Reprinted from the

Proceedings of the GCC Developers' Summit

June 28th–30th, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon Incorporated*
C. Craig Ross, *Linux Symposium*

Review Committee

Ben Elliston, *IBM*
Janis Johnson, *IBM*
Mark Mitchell, *CodeSourcery*
Toshi Morita
Diego Novillo, *Red Hat*
Gerald Pfeifer, *Novell*
Ian Lance Taylor, *Google*
C. Craig Ross, *Linux Symposium*
Andrew J. Hutton, *Steamballoon Incorporated*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Matrix flattening and transposing in GCC

Razya Ladelsky
IBM Haifa Labs
razya@il.ibm.com

Abstract

The layout of data in memory can have a significant effect on the performance of applications. Several compilation techniques can be used to optimize this layout. This paper describes two such optimizations: the first is matrix flattening, whose purpose is replacing a m-dimensional matrix with its equivalent n-dimensional matrix, where n<m. The frequent case is when a multidimensional matrix is flattened to its equivalent one dimensional matrix. This reduces the level of indirection needed for accessing the elements of the matrix. The second optimization is matrix transposing, which swaps rows and columns, and by doing so improves cache locality.

Both optimizations are interprocedural, and use the TREE-SSA based interprocedural framework, currently on ipa branch. In this paper we describe the algorithms used, as well as implementation issues. Preliminary results show substantial improvements for some floating point benchmarks, and no degradation for others. Finally we discuss the current status, future work and potential extensions.

1 Matrix reordering—Motivation

In any processor that contains some sort of memory hierarchy, access to a close location is faster than to farther ones. Achieving high performance requires effective use of cached data, meaning cache locality should be exploited as much as possible. One way to achieve locality is to transform loop nests. There has been a lot of work in the area of loop transformations. Among the techniques used are unimodular and nonunimodular iteration space transformations, tiling, loop fusion, and affinity scheduling. These techniques focus on changing the iteration space traversal order, and by doing so, they indirectly improve cache locality. Data transformation, however, focus directly on the data space, by changing the data layouts for better locality to be achieved. Unlike loop transformations, data transformations are not constrained by data dependencies, and can be applied to imperfectly nested loops. Also, while loop transformations affect all the matrices referenced in the loop nest, data transformations do not.

Not much work has been done in the area of data transformations in GCC. In this paper we present two matrix layout transformations. First we present matrix flattening, which flattens multi-dimensional dynamic matrices into contiguous memory space to achieve better reference locality. The second optimization also flattens multi dimensional dynamic matrices into one dimensional matrices, but reorders the elements of the flattened matrix differently, i.e. not corresponding to the original dimensional organization.

2 Flattening a matrix

Throughout this paper, we'll refer to the dimensions of matrices as inner/outer, lower/higher. For a matrix m[i1][i2]...[ik], we refer to i1 as the outer most dimension, and ik as the inner most. It is also convenient to number them. We'll number i1 as dimension 0, i2 as dimension 1, and so on. A lower dimension means an outer one.

2.1 Flattening—the idea

In order to explain the matrix flattening idea, we'll look at a two dimensional dynamically allocated matrix, `a` (as defined in C language). Let's denote the outer dimension as dimension 0, with size N, and the inner one as dimension 1, of size M. A typical creation of `a` will be:

```
a = (int**) malloc (N)
for (i=0; i<N; i++)
  a[i] = (int *) malloc (M);
```

Let's assume M=3 for simplicity of the example. The layout for this matrix organization is described in Figure 1:

```
a[0] = {x0, y0, z0}
a[1] = {x1, y1, z1}
...
a[N-1] = {xN-1, yN-1, zN-1}
```

Figure 1: two dimensional matrix example

Each access to an element of the matrix involves two levels of indirections. An access to a[i][j] requires accessing to a[i] in order to load the address of the inner dimension, and then accessing the j-th element of that dimension. Flattening the matrix `a` means that we allocate only one memory space (of size N*M), and place the elements contiguously in it. The matrix becomes one dimensional. An access to a[i][j] will be translated to accessing `base_of_new_allocated_matrix + new_offset`. The two references to memory that were required before, are replaced with only one memory access. If this access is in a loop, many memory references will be saved.

The resulting flattened matrix can be visualized as:

```
{x0 y0 z0 x1 y1 z1 ... xN-1 yN-1 zN-1}
{ dim 1 } { dim 1}     {    dim 1    }
{             dim 0                   }
```

An access to a[i][j] will now be an access to a[i*M+j].

We see that the elements are organized according to their original allocation, with the elements of the rows placed serially, and the rows placed one after the other in the order of iterating the outer dimension from 0 to N-1 (i.e. a[0], then a[1], and so on).

Flattening can be done for multiple dimension matrices whose dimension is greater than 2 as well. According to the same concept demonstrated for a 2 dimensional array, the elements of the inner most dimension are laid out serially. Iterating the outer dimension serially determines the order of the inner dimension, and so on, up to the outer most dimension. For example, given the the matrix in Figure 2:

The flattening can be visualized as:

{x0 x1 y0 y1 z0 z1 v0 v1 w0 w1 s0 s1}

We'll look at this example again in Section 6.4 that deals with transposing, and we'll see that there are alternative flattened layouts.

2.2 Partial flattening

In some cases the whole matrix cannot be flattened. This happens if some of its dimensions escape the application. Since flattening

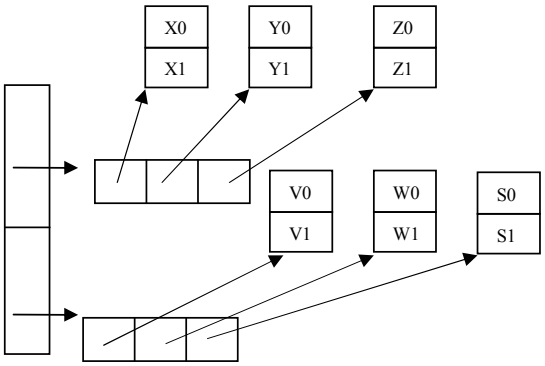

Figure 2: three dimensional matrix example

changes the definition of the matrix, we need to see all uses of the dimensions we flatten. Therefore, the escaped dimensions can't be flattened.

We'll present escape cases using an example of a three dimensional matrix, a[M][N][L]. The escape cases are:

1. The matrix, or part of it, is passed as an argument, e.g. call func(a[i][j]) or call func (&a[i][j][k]) causes the innermost dimension to escape.
2. part of the matrix is modified: a[i][j] = x; Again, the last (inner most) dimension escapes.
3. Multiple allocations for the same dimension. This is actually a private case of assigning a value to part of the matrix (case 2).

We call the outer most dimension that escapes, the `minimum indirection level`, which we'll also mark as `min_escape_level`, as any lower dimension (lower == outer) doesn't escape, and is safe for transformation, while any higher dimension is considered escaping. The dimensions from 0 (outer most) to the minimum indirection level are considered safe for transformation, because we know their behavior, while the rest should be left unchanged.

For the above escape examples of matrix `a`, `a`'s minimum indirection level is 2, meaning that dimensions 0,1 are safe for transformation and will be flattened, while the inner most dimension remains unchanged. Therefore, an access of a[i][j][k] to the old matrix, will be replaced by a[i*M+j][k] of the new partially flattened matrix.

3 Implementation Overview

Matrix flattening and transposing code has been developed in the ipa branch. Matrix flattening and transposing is one of the IPA passes, and is enabled by the `-fmatrix-flatten` flag. The option `-fipa-dump-mreorg` dumps information related to the optimization. The matrix flattening and transposing code can be found in the file `matrix-reorg.c`. The implementation is divided into three parts:

Analysis stage – performs a local analysis of the method to collect information about the allocation sites and the access sites for each matrix. the information is recorded in the mi structure (described below).

Decision making stage – decides whether to flatten the matrix, or the transposed matrix.

Transformation stage – changes the allocation and access sites in the various functions that access the matrix.

3.1 Data Structures

We need to collect a lot of data regarding the allocation and accesses of the original matrix. Everything is stored in three main structures:

`matrix_info` stores matrix information. It contains the following fields:

actual_dims – Maximum number of indirections used to access the matrix.

min_indirect_level_escape – Minimum indirection level that escapes. 0 means that the whole matrix escapes, k means that dimensions k to actual_dims escape.

malloc_for_level – Holds the allocation site for each level (dimension).

allocation_function_decl – The location of the allocation sites (all allocations must be in one function only)

free_info – The calls to free at each level of indirection.

dimension_size – An array holding the size for each dimension.

dim_hot_level – Array representing the hotness of each level, for transposing decision.

access_l – An array of the access sites of the matrix.

dim_map – A mapping from the old dimensions to their new order in the flattened matrix (also used for transposing)

`access_site_info` stores information about matrix access sites. It contains the following fields:

stmt – The access statement

offset – In case it is a `PLUS_EXPR`, this is the offset.

level – The level of indirection of this access statement.

iterated_by_inner_most_loop – Used for deciding whether to flatten the matrix or the transposed matrix.

`allocation_info` structure mainly contains *stmt*, which is the allocation statement.

3.2 Matrix reorg functions

The driver of matrix reorg is `matrix_reorg()`. The analysis part of matrix reorg is implemented by the following functions:

`analyze_matrix_allocation_site()` – Performs analysis of the allocation sites: recognizes the various dimensions' allocations and records them in the allocation site structures of the matrix.

`analyze_matrix_accesses()` – Recognizes and records the access sites of the matrix. By analyzing the accesses, it determines and records the minimum escape level and the actual dims of the matrix (the maximum level the matrix is accessed at).

`analyze_transpose()` – Decides whether to flatten the matrix as it is or flatten a transposed matrix. Determines the permutation of dimensions in which the matrix will be laid out.

The transformation part of the optimization is implemented by the functions:

`transform_allocation_sites()` – Replaces multiple mallocs with the equivalent malloc for the flattened matrix.

`transform_access_sites()` – Changes the access sites of the matrix to access the new flattened (possibly transposed) matrix.

4 Analysis phase

The analysis part of the optimization collects information about the allocation and access sites of the matrix. In the process, it determines K, the escape level of a N-dimensional matrix (K <= N), that allows flattening of the external dimensions 0, 1,..., K-1. An escape level of 0 means that the whole matrix escapes and no flattening is possible. The analysis phase is divided into analyzing the allocation sites and analyzing the access sites. We'll demonstrate the analysis methods using ssa representation. Both analyses are recursive. The two triggering calls from the analysis driver are as described below:

```
for (i=0; i < num_ssa_names; i++)
{
 ssa_var = ssa_name (i)
 stmt = DEF_STMT (ssa_var)
 if rhs is a matrix decl
   analyze_matrix_accesses
     (ssa_var, (level=)0)
 if lhs is a matrix decl
   analyze_allocation_site
     (DEF_STMT(ssa_var), (level=)0)
}
```

`analyze_allocation_site (stmt, level)` – Given a statement whose left hand side is a matrix variable, we traverse backwards to find the definitions that reach this variable, until we get to the allocation site (malloc, calloc, etc.) If we get some other kind of definition, we mark the matrix escaping. The patterns we are looking for are demonstrated in Figure 3.

```
analyze_allocation_site(stmt, level) {
 if (code (stmt) != MODIFY_EXPR)
   mark_matrix_escaping (level)

 rhs = get right hand side of stmt
 if (code (rhs) == SSA_NAME)
   analyze_matrix_allocation_site(DEF_STMT(rhs))
 else if (code (rhs) == CALL_EXPR) {
   if (call ! memory_allocation
       && call ! memory_free)
     mark_matrix_escaping (level)
   else
     add allocation site if there isn't
     a prior allocation statement at
     this level
 }
}

/* If needed, update the minimum escape
   level of the matrix.  */
mark_matrix_escaping (level) {
 if (matrix->min_escape_level > level)
   matrix->min_escape_level = level
}
```

Figure 3: Analyze Allocation Sites Algorithm

`analyze_accesses (ssa_var, level)` – Given a ssa var that is related to the matrix, and level of indirection corresponding to the current access level, we determine whether the matrix escapes at that level. If not, we record this access with the appropriate level. We follow the uses of the `ssa_var`, and analyze what happens to them. The handling of various use cases is described in Figure 4.

5 Transformation phase

In this phase we define the new flattened matrices that replace the original matrices in the code. We need to transform the allocation and the access sites of the matrix.

```
analyze_accesses(ssa_name, level) {
 use_stmt = use_stmt (ssa_var)
 switch (code(use_stmt)) {
  case PHI:
    /* The statement is of the form:
       res = PHI <ssa_name, ...>  */
    check escaping levels of arguments of PHI:
    if not all have the same level of escaping
      mark_matrix_escaping
           (minimum of escaping levels)
      return
    else
      analyze_matrix_accesses (res, level)

  case CALL_EXPR:
    /* The statement is of the form:
       ... = call (ssa_name, ...)  */
    if (call != alloc or free)
      mark_matrix_escaping (level)
    else
      record alloc or free, dim = level

  case MODIFY_EXPR:
    if ssa_name in lhs {
      /* The statement is of the form:
         *ssa_name = rhs */
      if rhs != ssa var
        mark_matrix_escaping (level)
      else
        /* Analyze the ssa var definition. */
        def_stmt = DEF_STMT (rhs)
        analyze_allocation_site
                  (def_stmt, level +1)
    } else ssa_name in rhs {
      /* Several optional patterns:  */

      /* Form is  ... = *ssa_name  */
      record access site with dim = level
      level ++
      goto acc

      /* Form is ... = call (ssa_name, ...) */
      if (call != alloc or free)
        mark_matrix_escaping (level)
      else
        record alloc or free site, dim = level

      /* Form is ... = ssa_name + ...  */
      record access site with dim = level

      /* Form is ... = ssa_name  */
acc:  if (lhs is ssa var)
        analyze_matrix_accesses(lhs, level)
      else
        mark_matrix_escaping (level)
    }
 }
}
```

Figure 4: Analyze accesses

5.1 Transforming allocation sites

Given an allocation function, which is the function that contains all of the matrix's memory allocations (an allocation for each dimension), we need to modify the code such that all memory allocations of dimensions that should be flattened, will be replaced with a single memory allocation of the equivalent size.

We calculate and produce code that reflects a new symbolic size for each dimension of the flattened (part of the) matrix. The new size symbolizes the over-all size of all elements contained within this dimension. Each new dimension size is placed in a new global variable, which we'll annotate as Ti, so it could be read from all functions that use the matrix (these values are used when changing the access sites of the matrix). The new size for dimension 0 holds the over-all memory size that should be allocated for the flattened part of the matrix. In the same manner of creating global variables for the new dimension sizes, we need to keep a global variable for each original size of the original dimension. We'll denote it T_ORIGi for dimension i, and it is used by the access sites to calculate the new offset.

When we start the transformation, we already have information about the matrix we want to transform. We collect this information in the analysis phase. This is the input for our algorithm. The minimum escape level has already been determined, and so has the original size for each dimension.

We'll define the following symbolic structures for each dimension i, where `i=0`, ..., `min_escape_level-1`: (`min_escape_level-1` is the inner most dimension of the matrix that should be flattened):

`dim_size_orig[i]` holds the original size of the dimension i.

```
transform_allocation_sites {
 for all i = (min_escape_level-1 to 0) {
  add_new_global_variable T_ORIGi
  emit code : T_ORIGi =  dim_size_orig[i]

  dim_size_orig [i] = Ti
 }
 prev = type_size[min_escape_level-1]
 for all i = (min_escape_level-1 to 0) {
  dim_num =
    dim_size_orig[i]
                / type_size[i]
  dim_size =
    prev * dim_num

  add_new_global_variable Ti
  emit code : Ti =  dim_size

  prev =  dim_size [i] = Ti
 }

 Remove all allocation statements for
 dimensions (1,...,min_escape_level-1)
 from the allocation function

 Change the allocation statement of level 0
 to allocate size according to T0.
}
```

Figure 5: Transform Allocation Sites Algorithm

`dim_size[i]` is initialized similarly to `dim_size_orig[i]`. This value changes throughout the algorithm to reflect the new symbolic dimension size.

`type_size[i]` holds the original size of the type of elements in dim i.

The algorithm is introduced in Figure 5.

The algorithm's output: for each dimension `i`, where `i=0,...,min_escape_level-1`, `dim_size[i]` and `Ti` contain the new symbolic dimension size for dimension `i`. `Ti` was inserted to the allocation function and can be read by other functions that access the matrix. All allocations of the dimensions that need to be flattened were replaced by a single equivalent memory allocation.

In other words, for a matrix of N dimensions, where we flatten dimensions 0 to D, the matrix's allocation sites:

```
malloc (dim (0))
malloc (dim (1))
 ...
malloc (dim (D))
malloc (dim (D+1))
...
malloc (N-1)
```

are transformed to

```
malloc (dim(0) * dim(1) * ...
        * dim (D) * size_type (D))
malloc(dim(D+1))
....
malloc (N-1)
```

The handling of free statements is simple: we remove all statements that free allocated memory for dimensions that we flattened, except for the free of the outer most flattened dimension.

5.2 Transform Access sites

Since the definition of the matrix has changed, it is obvious that the accesses to the matrix should be modified. We need to calculate the new offset from the base address of the new flattened matrix. For matrix `a` of k dimensions of sizes D1...Dk, and whose escape level is m<=k, we'll denote `a'` as the new allocated matrix. The original access a[I1][I2]...[Ik] will be translated to:

b[I(m+1)]...[Ik]

where

b = a' + I1*D2...*Dm + I2*D3...*Dm + ... + Im

In other words, access a[I1][I2]...[Ik] is translated to:

a'[I1*D2...*Dm + I2*D3*...*Dm + ...
+ Im][I(m+1)]...[Ik]

In the analysis phase, we determined the level of each access site, which is the dimension of the matrix accessed by this access site. We need to change only accesses whose level is lower (outer) than the matrix's escape level, because these are the accesses to the dimensions that were flattened.

The algorithm is introduced in Figure 6. We'll refer to the same structures described in the previous subsection.

```
transform_matrix_accesses {
 level = access -> level
 while (access in access list) {
   if (level <= min_scape_level)
     continue;
   if (access->stmt includes offset) {
     num = access->offset / type_size [level]
     dim_num =
       dim_size_orig[level] / type_size [level]

     new_offset =
       num *
         (dim_size[level] / dim_num)

     replace offset with new_offset
   }
   if (access->stmt includes dereferencing
       && acc_info->level < min_escape_level-1)
          remove the stmt from code
 }
}
```

Figure 6: Transform Accesses Algorithm

Notice that `dim_size[level]` was assigned with the global variable Tlevel and `dim_size_orig[level]` was assigned with the global variable `T_ORIGlevel`. (These global variables were defined in the allocation function, see `transform_allocation_sites` algorithm in the previous subsection).

The value `dim_size[level] / dim_num` represents the size of the dimension at `level + 1`.

The output of the `transform_accesses` algorithm is the code created for the new offsets and the removal of accesses to flattened dimensions.

6 Matrix Transpose

6.1 Transposing—the idea

If we look at the order in which we organized the flattened matrix, we notice that the elements of the inner most dimension were placed consecutively. For example, for a two dimensional matrix `a`, the elements of the array a[0] were placed one after the other, then the elements of a[1], and so on. This could produce good cache behavior if the elements of the inner dimension are accessed sequentially by the program. For example:

```
for (i=0; i<N; i++)
  for (j=0; j<M; j++)
    access to a[i][j]
```

However, if the accesses are of the following form:

```
for (i=0; i<N; i++)
 for (j=0; j<M; j++)
   access to a[j][i]
```

The outer dimension is iterated sequentially. We basically iterate the columns and not the rows. Therefore, if we placed the elements of the columns serially, and the columns one after the other, we would have had an organization with better (cache) locality. Another way of looking at it is that if the matrix a was organized in a column-major fashion, we would have had better data locality. Flattening this column-major matrix could produce even better performance.

In order to achieve this effect, we conceptually transpose the matrix `a` (i.e. swap the columns and rows), and then flatten it. For example, the row-major matrix `a` in Figure 1 was flattened in Section 2.1. Now we show the flattened `transposed` matrix:

```
{x0 x1...xN-1 y0 y1...yN-1 z0 z1...zN-1}
{dimension 0} {dimension 0}{dimension 0}
{           dimension   1              }
```

dimension 0 was originally the outer dimension, and dimension 1 was the inner one. When organizing the transposed layout, dimension 0 is treated as the inner dimension, and dimension 1 as the outer one.

This can be enhanced for multiple dimensioned matrices as well (the transformation for a three dimensional matrix is exemplified in the further subsections).

6.2 Transposing - transformation phase

The decision making phase explained in the next subsection, determines whether to flatten the `transposed` matrix. It then supplies a mapping of the dimensions, which we'll denote dim_map. It is a function that maps the dimensions according to their new order in the flattened matrix. It is basically a permutation of the dimensions. The mapping for the example we saw in the above subsection, will be:

```
dim_map[0] = 1
dim_map[1] = 0
```

For dimensions i=0,1,...,k, dim_map[k] represents the dimension that will be treated as the innermost dimension, dim_map[0] represents the outermost.

The transformation part of matrix flattening which was shown in Figure 5 was enhanced to work with this dim_map, in order to enable flattening of transposed matrices as well.

```
transform_allocation_sites {
 for all i = (min_escape_level-1 to 0) {
   add_new_global_variable T_ORIGi
   emit code : T_ORIGi =  dim_size_orig[i]

   dim_size_orig [i] = Ti
 }
 prev = type_size[min_escape_level-1]

 for all i = (min_escape_level-1 to 0) {
   dim_num =
     dim_size_orig[dim_map[i]]
                 / type_size[dim_map[i]]
   dim_size =
     prev * dim_num

   add_new_global_variable Ti
   emit code : Ti =  dim_size

   prev =  dim_size [dim_map[i]] = Ti
 }

 Remove all allocation statements for
 dimensions (1,...,min_escape_level-1)
 from the allocation function

 Change the allocation statement of level 0
 to allocate size according to T0.
}
```

Figure 7: Transform Allocation Sites - complete algorithm

For flattening the matrix without transposing, the mapping is simply the identity function. Note that only the allocation sites transformation needs to interact with the dim_map. Once it assigns the structure dim_size (which holds the size of each dimension in the new flattened organization), the access sites transformation already looks at updated dimension sizes structure, which actually identifies how the allocation is done.

The algorithm flattens the matrix while organizing the elements in the order determined by the dim_map. The algorithm is demonstrated in Figure 7.

6.3 Transposing—decision-making phase

As we've seen, the profitability of flattening the transposed matrix depends on the accesses to

the matrix. If all, or most accesses to the matrix are of the form

```
for (i=0; i<N; i++)
 for (j=0; j<M; j++)
   access to a[j][i]
```

then it would be profitable to flatten the transposed matrix. However, if all or most accesses are of the from

```
for (i=0; i<N; i++)
 for (j=0; j<M; j++)
   access to a[i][j]
```

then it would be more valuable to flatten the matrix as it is. In order to calculate the profitability, we collect two types of information regarding the accesses:
1. profiling information used to express the hotness of an access, that is how often the matrix is accessed by this access site (count of the access site).
2. which dimension in the access site is iterated by the inner most loop containing this access.

The matrix will have a calculated value of weighted hotness for each dimension. Intuitively the hotness level of a dimension is a function of how many times it was the most frequently accessed dimension in the highly executed access sites of this matrix.

As computed by following equation:
m n
$\sum\sum dim\_hot\_level[i] +=$
j i
acc[j]->dim[i]->iter_by_inner_loop * count(j)

Where n is the number of dims and m is the number of the matrix access sites.

The organization of the new matrix should be according to the hotness of each dimension. The hotness of the dimension implies the locality of the elements.

6.4 Flattening transposed matrix - example

Let's look at the three dimensional matrix in Figure 2:

Let's assume that there are two access sites to this matrix.

```
access site 1:
 for (iterate i)
  for (iterate j)
    for (iterate k)
      mat [i][j][k]

access site 2:
 for (iterate i)
  for (iterate j)
    for (iterate k)
      mat [k][i][j]
```

and `count(access site 2) > count(access site 1).` According to the decision making function, we have:

access site 1: the inner most dimension (dimension 2) is iterated in the inner most loop,

dim2->iter_by_inner_loop == 1 .
access_site 1 -> dim_hot_level [0] = 0
access_site 1 -> dim_hot_level [1] = 0
access_site 1 -> dim_hot_level [2] = count (access_site 1)

access site 2: dimension 1 is iterated in the inner most loop
access_site 2 -> dim_hot_level [0] = 0
access_site 2 -> dim_hot_level [1] = count (access_site 2)
access_site 2 -> dim_hot_level [2] = 0

Summing up the two access sites produces:
dim_hot_level [0] = 0
dim_hot_level [1] = count (access_site 2)
dim_hot_level [2] = count (access_site 1)

Since count(access site 2) > count(access site 1), the hottest dimension is dimension 1, then dimension 2 and lastly, dimension 0.

The resulting dim_map will be:
dim_map [0] = dimension_0
dim_map [1] = dimension_2
dim_map [2] = dimension_1

and the resulting flattened matrix:

{x0 y0 z0 x1 y1 z1 v0 w0 s0 v1 w1 s1 }

7 Issues

We'll discuss a few algorithmic and design issues related to matrix flattening and transposing optimizations.

7.1 Alignment issue

Transforming the allocation of the matrix might cause a misalignment issue that did not exist before the flattening. For example, flattening a 2 dimensional matrix according to the algorithm previously introduced, results in placing row after row sequentially. Originally these rows were created by separate memory allocations, which guaranteed their alignment. For optimizations that follow matrix flattening, there is no correctness problem because they already "see" the new flattened matrix. However, problems may occur for optimizations that precede matrix flattening. One such example is manual vectorization. Vector operations may take into account that the rows are aligned. The new matrix layout does not maintain the alignment for the rows anymore. This may be problematic and cause crashes. There are several possible solutions. One is padding between the rows. Another is to disable the optimizations if such vector operations exist.

We chose not to insert padding for two reasons: first is that it could effect cache behavior, and secondly is that it gets very complicated when we flatten a transposed matrix.

Therefore, the matrix flattening optimization should be disabled when vector operations exist. We disable the optimization if any of the following exist in the code:

1. convert expression of vector type.
2. call to builtin function which gets the matrix (or part of it) as an argument.
3. ASM operations.

7.2 Fortran matrices

The matrices in fortran are already flattened by the front end. Therefore matrix flattening is useless for Fortran matrices. However, transposing can be useful for these matrices. The flattened matrices need to be recognized and then reordered in a more optimal organization. This is currently not handled.

7.3 Whole program

Matrix flattening requires whole program view, as we change the definition of the matrices, and therefore all uses must be seen by the analysis and transformation. Therefore, we apply the optimization only if whole program flag is enabled.

8 Status

Matrix flattening code was submitted to ipa branch in March 2006. It includes the capability to flatten dynamically allocated non-escaping C matrices. The code produced improvements of 35% on art and 9% on equake (tested on linux powerpc).

The code for transposing is currently being developed, and preliminary results show a 200% improvement for transposing the matrices in art. The decision making phase has not been completed. It includes using the profiling information and also gathering the information about which dimension was iterated by the inner most loop at each access. This work will be submitted shortly.

Currently the code handles only dynamically allocated C matrices. Future enhancements include enhancing this work for other types of matrices: statically defined matrices, Fortran matrices, and so on. Future work also includes extending the patterns recognized by the analysis phase, and also enhancing the reordering algorithms for various matrix layouts.

9 Acknowledgements

I would like to thank Revital Eres and Mustafa Hagog, who wrote the preliminary versions of matrix flattening code, which I continued developing.

I would like to thank Jan Hubicka, Daniel Berlin and Sebastisn Pop who advised with design and implementation issues.

I would like to thank Peter Bergner for reviewing this paper, the IBM Haifa team for helpful discussions, and all GCC contributors who offered help and comments.

References

[1] M. Cierniak, W. Li. *Unifying data and control transformations for distributed shared memory machines*. ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.

[2] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. *A Matrix-Based Approach to Global Locality Optimization* . Parallel Architectures & Compilation Techniques (PACT'98).

[3] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, J. Ramanujam. *A Data Layout Optimization Technique Based on Hyperplanes* . Proceedings of the Eleventh International Parallel Processing Symposium, 1997.

[4] M. Kandemir, A. Choudhary, J. Ramanujam, N. Shenoy, P. Banerjee *Enhancing Spatial Locality Via Data Layout Optimizations* . In Workshop on Automatic Parallelisation, Southhampton, UK, Sept. 1998.

[5] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[6] Ken Kennedy, Randy Allen *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann; 1st edition (October 22, 2001)