Lecture Notes on
Linear Cache Optimization & Vectorization

15-411: Compiler Design
André Platzer

Lecture 25
November 18, 2010

1 Introduction

The big missing questions on cache optimization are how and when generally to transform loops? Is there a big common systematic picture? How to get fast by vectorizing and/or parallelizing loops after the loop transformations have made some loops parallelizable? And, finally, how can we use more fancy transformations for complicated problems.

2 Linear Loop Transformations

We have seen a number of loop transformations, but they all have been different, needing different analysis and implementation. However, a closer look reveals that the previous list of loop transformations (permutation, reversal, skewing) all follow a general pattern of linear loop transformations. Each of those transformations (and combinations and many others) can be represented by unimodular linear transformations. That is, such a transformation on n loops corresponds to an $n \times n$ integer matrix $U \in \mathbb{Z}^{n\times n}$ with determinant $\det U = \pm 1$. Because of the unit determinant $\det U$, they actually form a group, because we can form inverses

$$GL(n,\mathbb{Z}) := \{U \in \mathbb{Z}^{n\times n} \ : \ \det U \in \{1,-1\}\}$$

These linear transformations are volume-preserving. This makes intuitive sense. After all, if the volume changes during a transformation, then the number of grid points in it changes too, which would change the total number of iteration points of the loop incorrectly. They also preserve the orientation if $\det U = 1$, otherwise orientations change.

For instance, here are the unimodular transformations for loop permutation (matrix U_p) and loop reversal (matrix U_r) and loop skewing by factor f (matrix U_f):

$$U_p = \begin{pmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & 0 & 1 & & & \\ & & & 1 & 0 & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{pmatrix} \qquad U_r = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & -1 & & & \\ & & & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix}$$

$$U_f = \begin{pmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & 1 & f & & & \\ & & & 0 & 1 & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{pmatrix}$$

The corresponding inverse transformations are

$$U_p^{-1} = U_p \qquad U_r^{-1} = U_r \qquad U_f^{-1} = U_{-f}$$

Since unimodular transformations form a group, every combination of any number of the above matrixes is again a unimodular transformation.

Given a unimodular matrix $U \in GL(n, \mathbb{Z})$, we can apply the corresponding transformation by multiplying the iteration vectors by U, which may swap loops and transform their bounds as in the previous examples. Using vectorial notation for the iteration vector i, linear loop transformation by the unimodular matrix U can turn

```
for each vector<int> i in order $\prec$ do
    A[Mi+c] = A[Ni+e] + 55
```

(for matrix M, N and vector c, e that correspond to the array accesses) obviously into

```
for each vector<int> j in order $\prec$ do
    k = $U$j
    i = $U^{-1}$k
    A[Mi+c] = A[Ni+e] + 55
```

because $U^{-1}U = UU^{-1} = id$. In particular, $U^{-1}Uj$ indeed equals j, hence has the same value as the original iteration variable that we previously called i. Now, if we make sure that we actually change the perfectly nested loops so that they directly iterate over $k = Uj$ instead of j (e.g., by swapping/reversing/skewing according to U), then we can use essentially copy propagation to reach the following result of linear loop transformation:

```
for each vector<int> k in order $U \prec$ do
    A[M$U^{-1}k$+c] = A[N$U^{-1}k$+e] + 55
```

Thus the overall effect of the linear loop transformation is to apply transformation U to the loops and make up for that by multiplying all uses of the induction vector by U^{-1}.

This linear loop transformation is admissible if, for all iterations $i, i' \in \mathbb{Z}^n$:

$$i\delta i' \quad\Rightarrow\quad Ui \prec Ui'$$

That is, whenever there is a data dependency between i and i', then, after the transformation U, the transformed Ui should come before the transformed Ui' in the iteration order.

3 SIMD Vectorization and SSE / MMX

For more information see Chapter 4 of http://www.intel.com/Assets/PDF/manual/248966.pdf. Vectorization turns a series of sequential instructions operating on scalars into a single instruction operating on multiple data (SIMD). Vectorization, of course, requires that the loop has been transformed with all previous techniques to make sure that all data dependencies are compatible with vectorization. This is essentially equivalent to the data dependency check for parallelization.

Intel's Streaming SIMD Extensions (SSE) require data to be aligned at addresses divisible by 16 bytes.

For instance, the following loop with 4 iterations

```
float *A, *B, *C;
for (int i = 0; i < 4; i++)
    C[i] = A[i] + B[i]
```

can be implemented in a vectorized form

```
  MOVAPS xmm0, A
  ADDAPS xmm0, B
  MOVAPS C, xmm0
```

this depends on knowing that A,B,C do not have other aliases. It also depends on knowing that the length of the arrays A,B,C is a multiple of 128bits. Otherwise either loop peeling can be used to handle the remainder or array padding to fill up the array with irrelevant 0 data.

Another consideration for using SIMD is that an array of structs is less useful than a struct of arrays, because, in a struct of arrays, the data of one field is layed out contiguously in memory, enabling SIMD processing. In contrast, an array of structs may have scattered access in memory.

Another thing that can be useful for SIMD computation is to use mask for implementing conditional effects per element in a single vector sweep:

```
short A[],B[],C[],D[],E[];
for (int i=0; i<N; i++)
    if (A[i] > B[i])
        C[i] = D[i]
    else
        C[i] = E[i]
```

```
  XOR eax, eax      ; SSE4.1 process 8 shorts at once
L:MOVQ xmm0,[A+eax]
  PCMPGTW xmm0,[B+eax]      ; gt compare mask
  MOVDQA xmm1,[E+eax]
  PBLENDV xmm1,[D+eax],xmm0
  MOVDQA [C+eax],xmm1
  ADD eax, 16
  CMP eax,N
  JLE L
```

4 Loop Sectioning / Section Striping

Loop sectioning is a simple transformation that turns a loop into two nested loops, where the inner loop traverses one section or block at a time. That is we turn a loop

```
for(int i = 0; i < N; i++)
  S
```

into two loops, where the inner one iterates over blocks of size B

```
for(int b = 0; b < N; b+=B)
  for(int i = b; i < b+B && i◁N; i++)
    S
```

For SIMD it is useful to pick block size B to be the size of the 128bit chunk size. Then the inner loop can be turned into a SIMD vector instruction.

The inverse transformation is possible too (turn nested perfect loops into a single loop) and called loop product transformation.

5 Loop Fusion

If two loops have the same index range and no tricky data dependencies exist between the loops, then loop fusion can turn two sequential loops

```
for(int i = 0; i < N; i++) {
  B[i] = A[i] + C[i]
}
for(int i = 0; i < N; i++) {
  R[i] = B[i] * (D[i] + A[i])
}
```

into a single loop

```
for(int i = 0; i < N; i++) {
  B[i] = A[i] + C[i]
  R[i] = B[i] * (D[i] + A[i])
}
```

After fusion, the latter loop body can then be optimized to remove the array B altogether if it is dead afterwards

```
for(int i = 0; i < N; i++) {
  R[i] = (A[i] + C[i]) * (D[i] + A[i])
}
```

In combination with loop sectioning, that loop can further be turned into SIMD instructions that add A[i] to C[i] with a single vector instruction and then multiply the result to the result of vectorially adding D[i] to A[i] with a single vector instruction. Another pleasant effect of loop fusion here is that this is a cache optimization, because the same element A[i] will be loaded into the cache only once, decreasing cache misses by half for large N.

Generally, loop fusion can also have a bad effect on caches for independent arrays where a lot of extra data will suddenly need to be stored in the cache, possibly leading to unnecessary cache spilling.

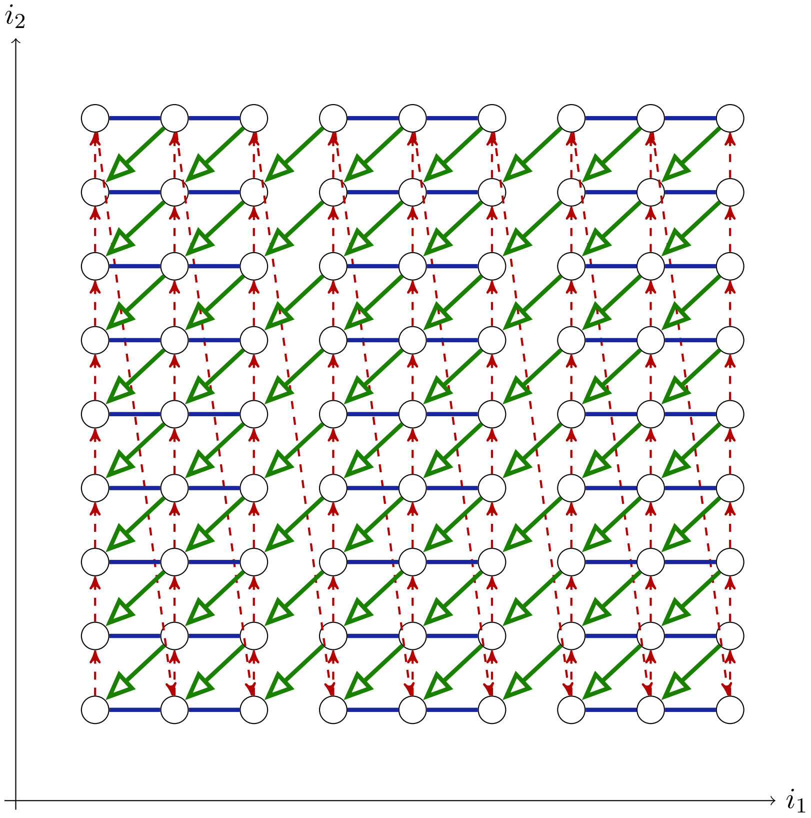
Bad data dependencies arise, e.g., if iteration i of the second loop already uses data like A[i+1] that the first loop writes in iteration $i + 1$ or if the second loop uses scalar data that the first loop defines, because the value of those scalars may be different after the first loop ran in full than in between.

Again, the inverse transformation is possible too and called loop splitting. Loop splitting can be useful to reduce the data load, possibly leading to reduced cache misses. It can help pulling parallelizable parts of a loop body out of the loop. Loop splitting can also be used to turn imperfectly nested loops into perfectly nested loops.

6 Loop Tiling

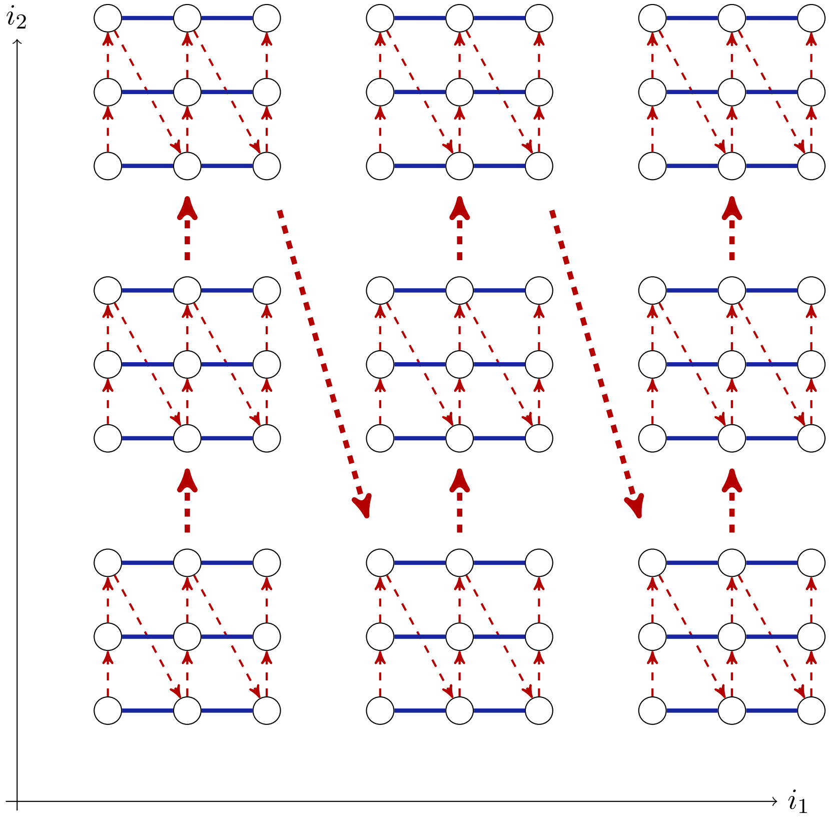
The loop blocking or loop tiling optimization partitions multidimensional loops into rectangles (or, more generally hypercubes), walking one rectangle at a time. This optimization can reduce cache capacity misses by making sure that the full cache line data within the rectangle will already be used before the data in the cache line is replaced by other information. That is useful if loop swapping doesn't solve the cache locality issues, e.g., because there are other operations that prevent it. In matrix multiplication, for instance, arrays are traversed in both column and row order, leading to bad cache effects. Loop tiling is an extremely useful optimization for matrix multiplication and similar problems of mixed array iteration.

```
for (i1=1; i1<=n; i1++)
  for (i2=1; i2<=n; i2++)
    A[i1,i2] = A[i1-1,i2-1] + 2
```



Dependency distance d=(1,1)
Lots of cache misses for large n
Swapping doesn't solve this problem

```
for (B1=1; B1<=n; B1+=3)      // loop tiling
  for (B2=1; B2<=n; B2+=3)    // loop tiling
    for (i1=B1; i1<B1+3; i1++)
      for (i2=B2; i2<B2+3; i2++)
        A[i1,i2] = A[i1-1,i2-1] + 2
```



After loop tiling, the loops iterate one block tile at a time
This simple loop tiling assumes that n is divisible by block size 3

As a very useful application of loop tiling, consider, for instance, matrix multiplication, which has both column and row traversal so that no loop swapping helps:

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    for (k=0; k<n; k++)
      R[i][j]  = R[i][j] + A[i][k] * B[k][j]
```

If all data fits into the cache and there are no problems with small associativity, then the innermost k loop may run fast because there are almost no cache misses (only once per cache line). If the matrix is too large then the data will have been flushed from the cache already before it's used next.

Loop tiling with a constant c that is just large enough for all the $c \times c$ matrix blocks to fit into the cache turns matrix multiplication into:

```
for (B=0; B<n; B+=c)      // loop tiling
  for (C=0; C<n; C+=c)    // loop tiling
    for (i=B; i<B+c&&i<n; i++)
      for (j=C; j<C+c&&j<n; j++)
        for (k=0; k<n; k++)
          R[i][j] = R[i][j] + A[i][k] * B[k][j]
```

The innermost R[i][j] access will be a cache hit every time but once. Nevertheless, it is an almost loop-invariant expression for the innermost loop k. Its address arithmetic is loop-invariant and can be moved out by loop-invariant code motion. Yet R[i][j] itself is not loop-invariant. After all it's assigned to all the time. One step better, however, we can even replace the assignment to R[i][j] by a scalar accumulator (favorably placed in a register as a very busy expression).

```
for (B=0; B<n; B+=c)     // loop tiling
  for (C=0; C<n; C+=c) // loop tiling
    for (i=B; i<B+c&&i<n; i++)
      for (j=C; j<C+c&&j<n; j++) {
        a = R[i][j]   // scalar optimization
        for (k=0; k<n; k++)
          s = s + A[i][k] * B[k][j]
        R[i][j] = s
      }
```

This also reduces the number of load/stores in the loop body to 2, which is good, because almost no architecture supports 3 load/stores very well.

References