Lecture Notes on
Linear Cache Optimization & Vectorization

15-411: Compiler Design
André Platzer

Lecture 25

1 Introduction

The big missing questions on cache optimization are how and when generally to transform loops? What is the best choice to find a loop transformation? 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 it contains inverses

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

Because of $|\det U| = 1$, these linear transformations are volume-preserving. This makes intuitive sense. After all, if the volume would change 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. Linear transformations furthermore preserve the orientation (here of the loop) 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 and thus represents a possible loop 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, the linear loop transformation by the unimodular matrix U can turn

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

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

```
for each vector<int> j in order ≺ do
    k = Uj
    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) so that k walks in the linearly transformed order $U\prec$, then we can use copy propagation to reach the following result of linear loop transformation:

```
for each vector<int> k in order U≺ do
    A[MU^{-1}k+c] = A[NU^{-1}k+e] + 55
```

Note that the matrix product MU^{-1} and NU^{-1} can be computed statically by the compiler and does not happen at runtime. 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 with U is admissible if, for all iterations $i, i' \in \mathbb{Z}^n$ and all data dependencies δ:

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

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 Data Dependency Analysis

What we need is a checkable criterion which we can use to decide if two array references lead to a dependency or not. In practice, we restrict our attention to affine array references in perfectly nested loops. That is, given two array accesses $A[Mi + c]$ and $A[M'i + c']$ we need to find instances $i, i' \in \mathbb{Z}^d$ of the iteration vector with $i \prec i'$ such that $Mi + c = M'i' + c'$. If such instances i, i' exist that have integer values and are within the loop bounds, then the two references have a data dependency. For perfectly

nesting forward iterating loops, the condition $i \prec i'$ on loop iteration order is defined as

$$(i_1, \ldots, i_d) \prec (i'_1, \ldots, i'_d) \quad \text{iff} \quad \exists k : i_1 = i'_1, \ldots, i_{k-1} = i'_{k-1}, i_k < i'_k$$

It is of slightly minor relevance, because a data dependency still exists even if $i = i'$, except that it is not loop carried. That is why it is sometimes ignored. Likewise, when $i' \prec i$, the condition $Mi+c = M'i'+c'$ results in an anti-dependency. Unfortunately, the problem of finding integer solutions of $Mi+c = M'i'+c'$ is that of solving linear Diophantine equation systems, which is NP-complete. The problem of finding integer solutions of $Mi+c = M'i' + c', i \prec i'$ is that of solving linear Diophantine inequality systems, which can still be solved with IP solvers and is NP-complete.

In order to find a reasonable approximation of the dependency analysis, we first pretend all references were dependent (which is a conservative overapproximation) and then remove some dependencies if we can show that they are independent. A simple (approximate) independency check is the gcd test. Suppose we have the loop

```
for (i=0; i<10; i++)
    A[m*i+c] = A[m'*i+c']+7
```

The question is, whether there are two positions of the iteration vector (here just $\check{i}, \hat{i} \in \mathbb{Z}^1$ for the write iteration $\check{i}$ and the read iteration $\hat{i}$) such that the array accesses interfere because $m * \check{i} + c = m' * \hat{i} + c'$.

$$\begin{aligned} & m * \check{i} + c = m' * \hat{i} + c' \\ \Leftrightarrow \ & m * \check{i} - m' * \hat{i} = c' - c \end{aligned}$$

The last linear Diophantine equation can only have an integer solution if $\gcd(m, m')$ divides $c' - c$. The reason for this is that, for integers $\check{i}, \hat{i}$, the term $m * \check{i} - m' * \hat{i}$ can only take on exactly the values that are multiples of the $\gcd(m, m')$. (More formally, this is a simple property of principal ideals in Euclidean domains.)

For instance, consider

```
for (i=0; i<10; i++)
    A[2*i+2] = A[2*i-2]+7
```

with the gcd check

$$\begin{aligned} & 2 * \check{i} + 2 = 2 * \hat{i} - 2 \\ \Leftrightarrow \ & 2 * \check{i} - 2 * \hat{i} = -4 \end{aligned}$$

We see that $\gcd(2, 2)$ divides -4. Thus, there could be a dependency. We read off by dividing both sides by the gcd 2 that $\check{i} - \hat{i} = -2$. This could be

1. A true dependency δ^t (read after write) of the form $\check{i} + 2 = \hat{i}$, i.e., with dependency distance 2. "The array position that we read at iteration $\hat{i}$ is the same one that we have written 2 iterations before at $\check{i}$."
   This dependency survives the check for loop bounds and is a real dependency.

2. And/or an anti-dependency δ^a (write after read) of the form $\hat{i} - 2 = \check{i}$. "The array position that we write at iteration $\check{i}$ is the same one that we have read 2 iterations before at $\hat{i}$." This dependency is incompatible with the constraint that $\hat{i} < \check{i}$, because we need data to use from previous iterations before we can use it for subsequent definitions.

More conservatively, one can consider all resulting dependencies as dependencies, without checking for feasibility with the loop bounds. Many more details and optimizations and accuracy improvements of the gcd test can be found in [Muc97, Sections 9.3,9.4].

More precise but computationally more involved algorithms exist, for instance, Fourier-Motzkin elimination for linear systems of inequalities.

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 or whatever size the vector instructions support. 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.

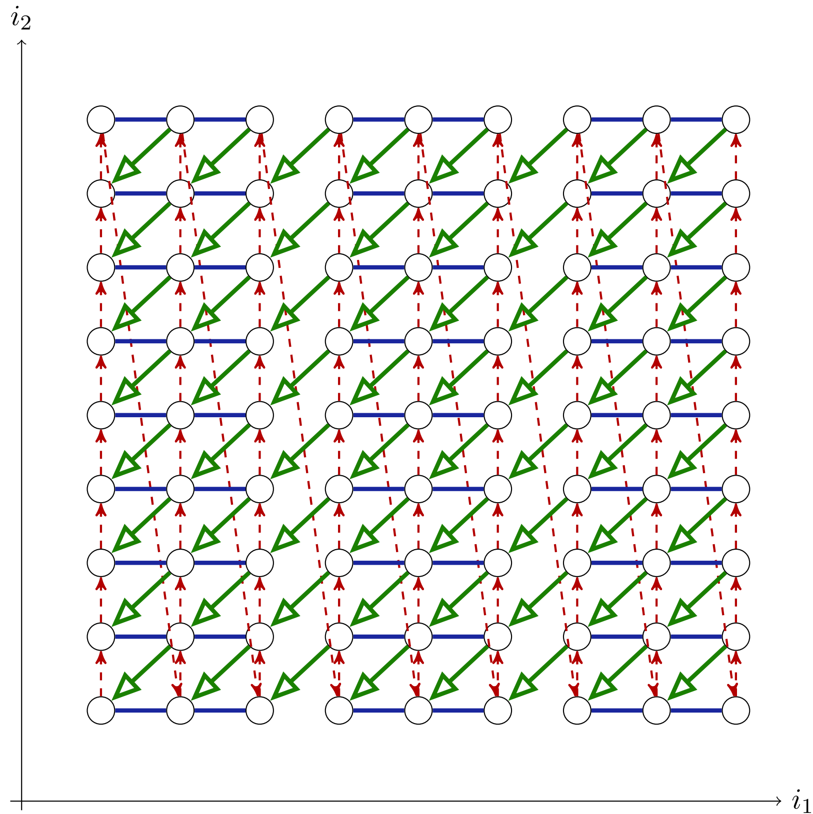
Again, the inverse transformation is possible too and called loop splitting. Loop splitting can be useful to reduce the data load, possibly leading to reduced cache misses. It can help pulling parallelizable parts of a loop

body out of the loop. Loop splitting can also be used to turn imperfectly nested loops into perfectly nested loops.

6 Loop Tiling

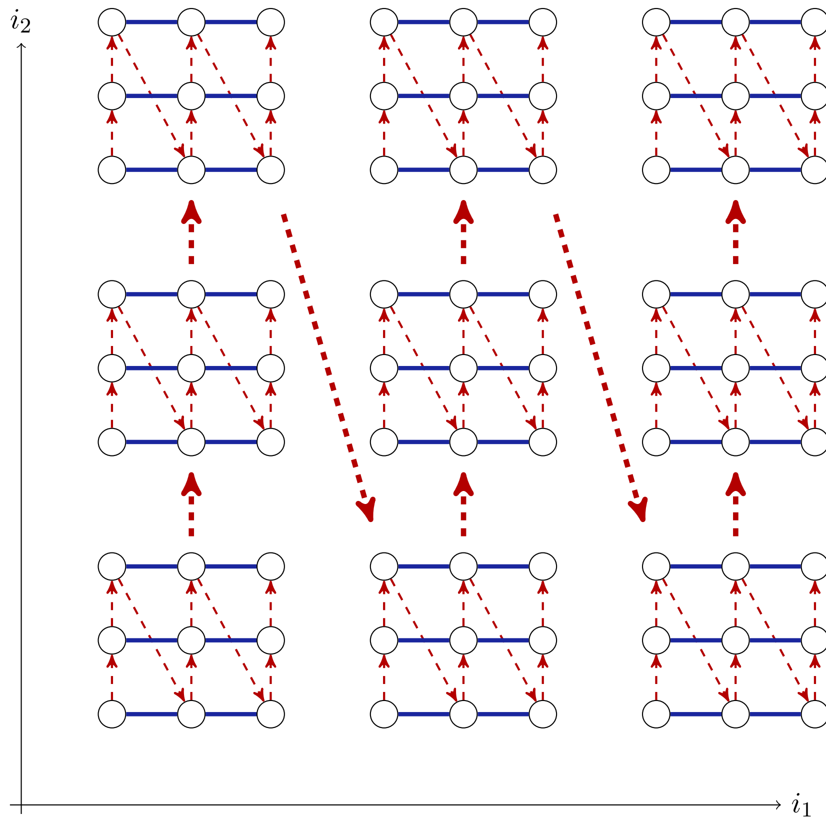
The loop blocking or loop tiling optimization partitions multidimensional loops into rectangles (or, more generally hypercubes), walking one rectangle at a time. This optimization can reduce cache capacity misses by making sure that the full cache line data within the rectangle will already be used before 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 regardless. 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
Otherwise use loop peeling
Loop tiling combines 2 loop sectioning and loop swapping

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.

Finally, strength reduction can be used to replace the respective address arithmetic by simple addition.

Quiz

1. How can you implement the gcd test efficiently and which approximations make most sense computationally?

References

[Muc97] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.