Lecture Notes on Loop Transformations for Cache Optimization

15-411: Compiler Design
André Platzer

Lecture 24

1 Introduction

In this lecture we consider loop transformations that can be used for cache optimization. The transformations can improve cache locality of the loop traversal or enable other optimizations that have been impossible before due to bad data dependencies. Those loop transformations can be used in a very flexible way and are used repeatedly until the loop dependencies are well aligned with the memory layout and cache effects are optimal. What is most important, however, is to keep track carefully under which circumstances the loop transformations are actually correct. We will pay attention to that. The same loop transformations are needed for loop parallelization and vectorization. For more information, refer to [Muc97].

2 Loop Permutation

Loop permutation swaps the order of loops. The purpose is to pull the loops out that actually carry the data dependencies, because the inner loops will then be parallelizable. Another purpose is to swap such that the inner loops traverse arrays according to the actual memory layout of the array to reduce cache misses. This can also help to enable optimizations with register use.

While irrelevant to the optimizations, we assume Fortran-style column major order for illustrations, where B[i1,i2] is adjacent to B[i1+1,i2] in memory. Consider

```
for (i1=0; i1<n1; i1++)
  for (i2=0; i2<n2; i2++)
    A[i1] = A[i1] + B[i1,i2]
```

From the $n1 * n2$ memory accesses to A[i1], we expect only n1 cache misses out of $n1 * n2$ access. Here A[i1] can even constantly remain in a register for n2 iterations of the inner loop. For B[i1,i2], however, the iteration order is non-local compared to the memory layout, hence we expect $n1 * n2$ cache misses for large data.

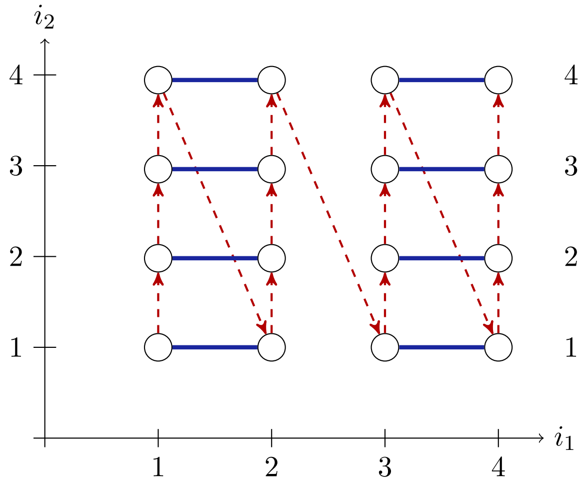
In contrast, when we swap loops

```
for (i2=0; i2<n2; i2++)    // loops swapped
  for (i1=0; i1<n1; i1++)
    A[i1] = A[i1] + B[i1,i2]
```

For A[i1], we expect $n1 * n2 * \frac{s}{cache\ size}$ cache misses, where s is the data size of the array elements, because we access $n1 * n2$ times non-locally. For B[i1,i2], we similarly expect only $n1 * n2 * \frac{s}{cache\ size}$ cache misses, because the loops traverse B locally along cache lines now.

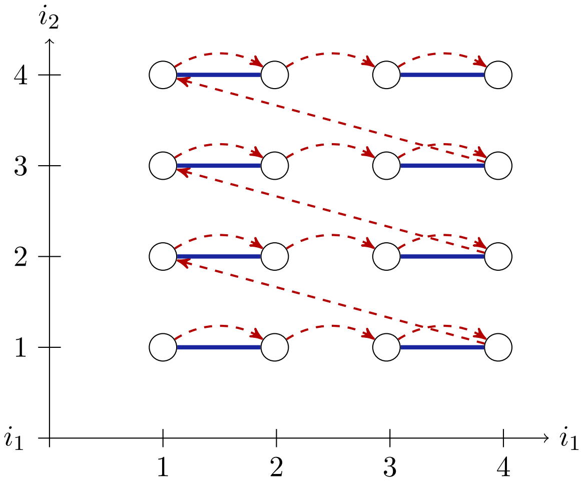
Loop permutation can have quite a remarkable effect.

```
for (i1=1; i1<=4; i1++)
  for (i2=1; i2<=4; i2++)
    A[i1,i2] = A[i1,i2] + 5
```



This loop iterates against the memory layout, which will lead to excessive cache misses on larger data.

```
for (i2=1; i2<=4; i2++) // swap
  for (i1=1; i1<=4; i1++)
    A[i1,i2] = A[i1,i2] + 5
```

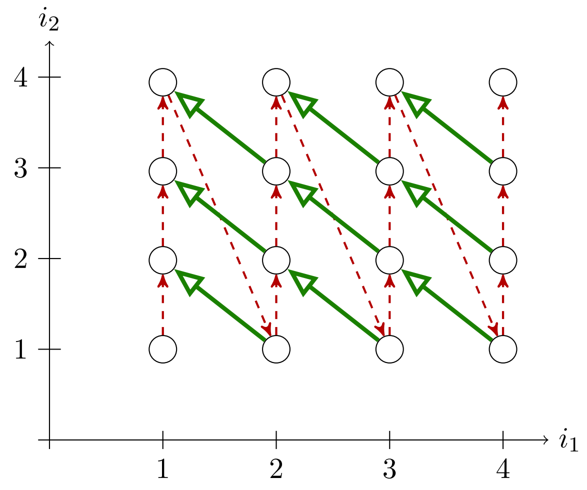


This loop iterates with the memory layout and uses the full cache line immediately.

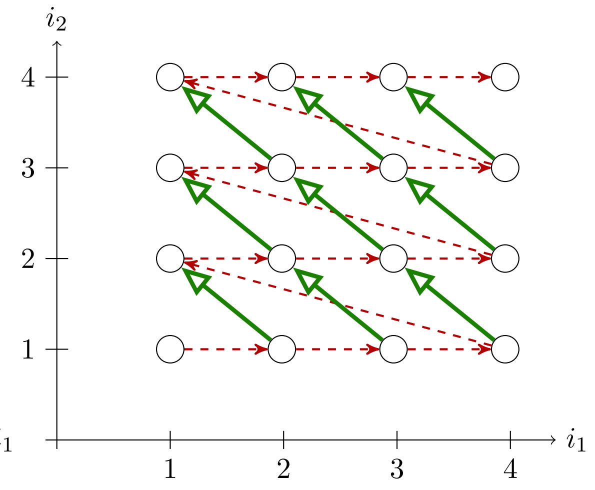
Loop permutation doesn't always have to be admissible, for instance, if we violate data dependencies by swapping loops. The dependency distance vector is permuted just like the iteration vector is. The important side condition is that the first non-zero sign of a dependency distance vector is not allowed to change.

```
for (i1=1; i1<=4; i1++)
  for (i2=1; i2<=4; i2++)
    A[i1,i2] = A[i1-1,i2+1] + 7
```

```
for (i2=1; i2<=4; i2++)   // swap
  for (i1=1; i1<=4; i1++)
    A[i1,i2] = A[i1-1,i2+1] + 7
```



Dependency distance d=(1,-1)



Dependency distance d=(-1,1)
Has different signs, thus loop swap illegal.

3 Loop Reversal

The point of loop reversal is to change the order in which a loop iterates. The primary advantage is that data dependencies change, which may enable other optimizations. Another advantage can be that the special "jump if zero" (JMPZ) machine instructions can be used when loops iterate down to 0.

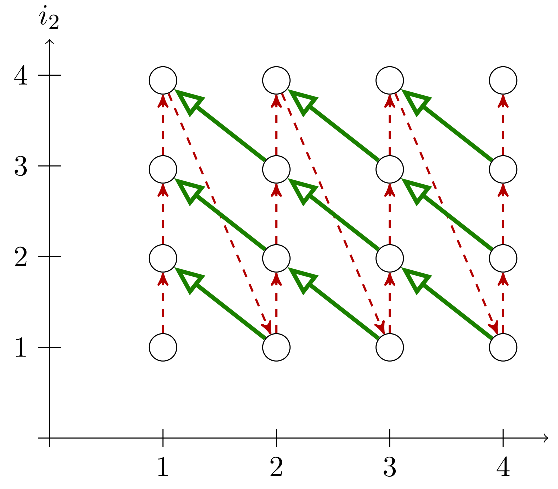
Loop reversal is admissible for loop q if all data dependencies are carried by outer loops, i.e.

$$\forall d \text{ dependency distance}: \ d_q = 0 \lor \ \exists k < q : \ d_k \neq 0$$

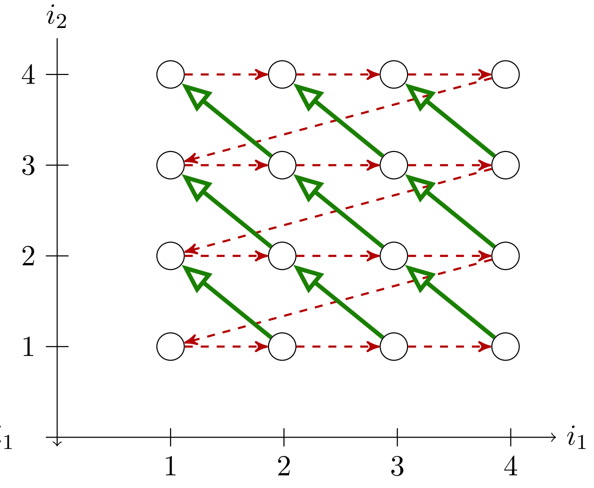
Otherwise, the direction of the dependency would be reversed by loop reversal. Take a look at the last loop permutation example again.

```
for (i1=1; i1<=4; i1++)
  for (i2=1; i2<=4; i2++)
    A[i1,i2] = A[i1-1,i2+1] + 7
```

```
for (i1=1; i1<=4; i1++)
  for (i2=4; i2>=1; i2--) // rev.
    A[i1,i2] = A[i1-1,i2+1] + 7
```



Dependency distance d=(1,-1)



Dependency distance d=(1,1)

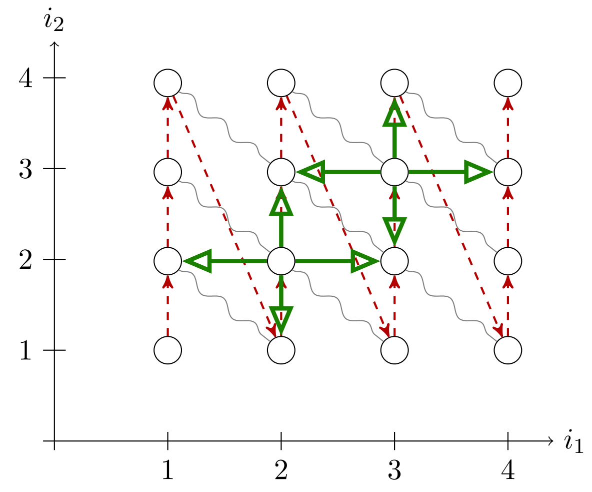
Note that, unlike loop i_2, the i_1 loop cannot be reversed, because that would lead to the bad dependency distance d=(-1,-1).

4 Loop Skewing

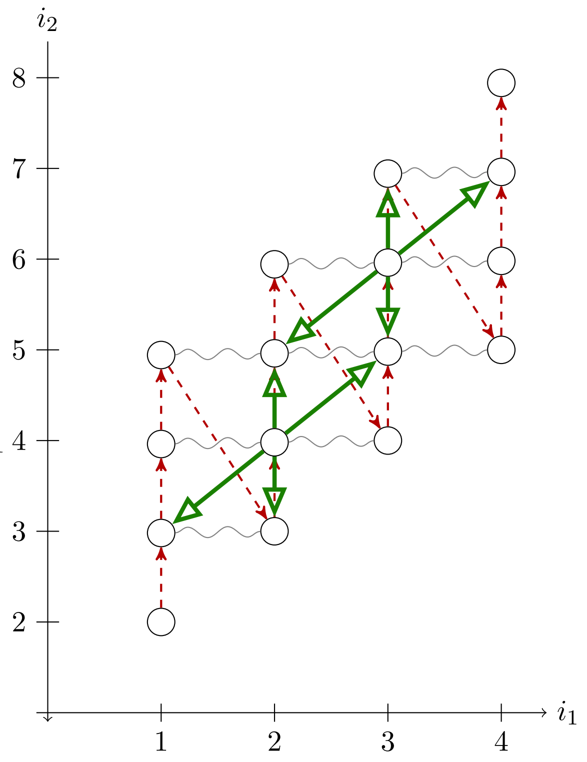
Loop skewing by a factor f adds $f * i_1$ to the upper and lower bounds of inner loop i_2 and makes up for that by subtracting $f * i_1$ again from all array accesses of i_2. The advantage of doing so is that the data dependency direction changes from d to $(d_1, f * (d_1 + d_2))$. By a smart choice of f, this change of the data dependency can enable other loop optimizations.

```
for (i1=1; i1<=4; i1++)
  for (i2=1; i2<=4; i2++)
    A[i1,i2] =
      (A[i1-1,i2]+A[i1+1,i2]
      +A[i1,i2-1]+A[i1,i2+1])/4
```

```
for (i1=1; i1<=4; i1++)
  for (i2=1+i1; i2<=4+i1; i2++)//skew
    A[i1,i2-i1] =
      (A[i1-1,i2-i1]+A[i1+1,i2-i1]
      +A[i1,i2-i1-1]+A[i1,i2-i1+1])/4
```



Dependency distances d=(1,0), d=(0,1).
Other dependencies d=(-1,0), d=(0,-1).
No loop can parallelize.
We cannot swap to fix this.



Loop skew by factor f=1
Dependency distances d=(1,1), d=(0,1).
Other dependencies d=(-1,-1), d=(0,-1).
Swapping is admissible and leads to
Dependency distances d=(1,1), d=(1,0).
Thus inner loop parallelizable.

```
for (i2=2; i2<=8; i2++)   //skew,swap makes i1 parallelizable
  for (i1=max(1,i2-4); i1<=min(i2-1,4); i1++)
    A[i1,i2-i1] =
      (A[i1-1,i2-i1]+A[i1+1,i2-i1]
      +A[i1,i2-i1-1]+A[i1,i2-i1+1])/4
```

Quiz

1. Why do many libraries for scientific computing feature representations of matrices as one-dimensional arrays? Discuss the tradeoffs for users and for compilers and for performance.
2. Develop a loop optimization that ignores cache optimization and just uses loop reversal so that JMPZ instructions can be used. Give all side conditions.
3. For each of the loop transformations in this lecture, give a natural example where that loop transformation increases cache efficiency. Give another natural example where that loop transformation would be incorrect.
4. Give a natural example where multiple loop transformations are needed to optimize cache efficiency.
5. Develop loop optimizations for the special case of loops that are uniform and have $c = 0$ for all array accesses $A[Mi + c]$? What are the side conditions in those cases? For what kind of programs are these special optimizations worth implementing?
6. In what kinds of natural examples does loop skewing play an important role?

References

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