Lecture Notes on
Cache Iteration & Data Dependencies

15-411: Compiler Design
André Platzer

Lecture 23

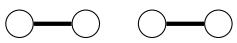
1 Introduction

Cache optimization can have a huge impact on program execution speed. It can accelerate by a factor 2 to 5 for numerical programs. Loops are the parts of the program that are generally executed most often. That is why cache optimization usually focuses exclusively on handling loops. Especially for loops that execute very often, optimizing small chunks of source code can have a fairly significant effect. Furthermore, loops often use mathematically regular access to arrays which is amenable to mathematical analysis. It turns out that the answers for cache optimization are exactly the questions that need to be answered for vectorization (into single-instruction multiple data) and parallelization optimizations. So all techniques we develop for one will help the others.

Some other information on cache optimization can be found in [App98, Ch 21].

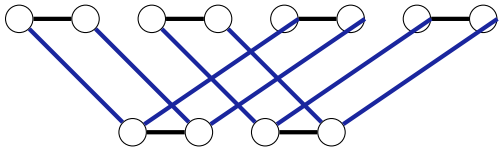
2 The Importance of Cache Optimization

For illustration purposes, take a look at a computer with a small cache of two cache lines with two data entries per cache line.



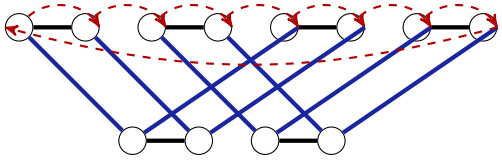
We further assume a directly mapped cache (without associativity) to simplify the presentation. We assume that a (small) array A with 8 elements has the following memory layout and maps as indicated by the solid blue

lines to the cache lines. We illustrate the cache to array field association in blue:



Cache Capacity Miss Consider the following loop that repeats the same data access in a one-dimensional array 8 times:

```
int A[8];
for(i = 0; i<8; i++)
  for(j = 0; j<8; j++)
    A[j] = ...
```

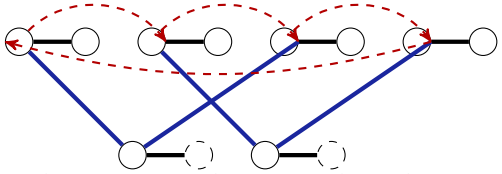


As illustrated by the dashed red iteration order, even though every array cell is used repeatedly, the data does not fit into the cache and will be reloaded twice during each repetition of the outer loop.

These 100% cache misses are caused by insufficient *cache capacity miss*. Hence, this loop should be cache-optimized to avoid suboptimal loop traversal in the cache.

Cache Line Capacity Miss Next, consider the following program

```
int A[8];
for(i = 0; i<8; i++)
  for(j = 0; j<8; j+=2)
    A[j] = ...
```

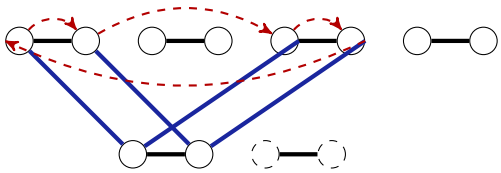


As illustrated by the dashed red iteration order, even though every array cell is used repeatedly, and even though the data would fit into the cache, unnecessary data wastes cache line space that is never used. Thus, the data will still be reloaded twice during each repetition of the outer loop.

These 100% cache misses are caused by insufficient *cache line capacity miss*. Hence, this loop should be cache-optimized to avoid suboptimal loop traversal in the cache. Here, possible cache optimizations for this particular loop traversal order also include reordering matrix elements, which is sufficient here. But that does not work if the program accesses the data in different ways in other parts of the program unless the compiler copies the matrix before. Thus, loop traversal optimization usually makes more sense.

Cache Conflict Miss Next, consider the following program

```
int A[2][4];
for(i = 0; i<8; i++)
  for(j = 0; j<2; j++)
    for(k = 0; k<2; k++)
      A[j,k] = ...
```



As illustrated by the dashed red iteration order, even though every array cell is used repeatedly, and even though the data would fit into the cache, the same cache lines are always used for all accessed data. Thus, the data will still be reloaded twice during each repetition of the outer loop.

These 100% cache misses are caused by *cache conflict miss*. Hence, this loop should be cache-optimized to avoid suboptimal loop traversal in the cache.

3 Data Dependencies

We use the following abbreviations for data dependencies between two locations ℓ and ℓ' in a program:

- $\ell \delta^t \ell'$ true data dependency (read after write)
- $\ell \delta^o \ell'$ output dependency (write after write)
- $\ell \delta^a \ell'$ anti-dependency (write after read)
- $\ell \delta^i \ell'$ input dependency (read after read)

These data dependencies can come in two flavors. Either just within a single iteration of the loop or they can be loop-carried, i.e., data dependencies between different loop iterations.

```
int A[8];
for(i = 1; i<8; i++) {
  ℓ1: A[i] = ...;
  ℓ2: x = A[i];          // ℓ1δ^t ℓ2 loop-independent
  ℓ3: y = A[i-1];        // ℓ1δ^t ℓ3 loop-carried
}
```

4 Loop Iteration Vectors

In the following we simplify the presentation by starting to use the language of linear algebra. We refer to previous lectures for guarding against array access out of bounds and mostly ignore this here. We generally assume that we have perfectly nested loops (outer loops have no other statements than just the induction variable increment and the inner loop).

```
for (i1 = o1; i1 < n1; i1++)
  for (i2 = o2; i2 < n2; i2++)
    for (i3 = o3; i3 < n3; i3++)
      ...
        for (id = od; id < nd; id++) {
          loop body;
        }
```

We assume that we have already performed induction variable analysis and found basic induction variables corresponding to the respective loop nesting. We denote the above loop iteration by a single iteration vector

$$i = \begin{pmatrix} i1 \\ i2 \\ i3 \\ \vdots \\ id \end{pmatrix}$$

Loops determine an iteration order $\prec$ on the index set $\mathbb{Z}^d$. We generally restrict attention to *affine array references*, i.e., those where the index expression is an affine linear function of the iteration vector i. That is all array accesses are of the form $A[Mi + c]$ for a matrix M and vector c. For instance,

$A[3{*}i_1{+}i_2{-}1, 4{*}i_2{+}5, 2{*}i_3]$ corresponds to access of A at $\begin{pmatrix} 3 & 1 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 2 \end{pmatrix} i + \begin{pmatrix} -1 \\ 5 \\ 0 \end{pmatrix}$

Two affine array accesses $A[Mi + c]$ and $A[M'i + c']$ are called *uniform* iff $M = M'$. If there is a dependency $L\delta^t L'$ between two statements writing to uniform array access $L : A[Mi + c] = ...$ and reading from uniform array access $L' : ...A[Mi + c']$ with $Mi + c \prec Mi + c'$ then the uniform distance $d := c - c'$ is called *dependency distance*. This distance only makes sense in the case of uniform access, because the difference is not a constant vector

otherwise. The data dependency vector says from which direction d data is needed to compute the new values. For example,

```
for (i1 = o1; i1 < n1; i1++)
  for (i2 = o2; i2 < n2; i2++)
    for (i3 = o3; i3 < n3; i3++)
      ...
        for (in = on; in < nn; in++) {
          A[M i + c] = A[M i + c'] + 5
        }
```

has dependency distance $d := c - c'$.

If loop i_k runs forward (e.g., via `ik++`) and the dependency distance is of the form

$$d = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ d_k \\ \vdots \\ d_n \end{pmatrix} \quad \text{for some } d_k > 0$$

then the loop i_k carries that data dependency, because the write into $A[Mi + c]$ writes into a memory location that will only be read at $A[Mi + c']$ later.

Loop i_j can be parallelized if the dependency distances d of all data dependencies satisfy

$$d_j = 0 \text{ or } d_k \neq 0 \text{ for some } k < j$$

Then such a dependency either has no data dependency ($d_j = 0$) or, if $d_k > 0$, depends on data from a past iteration of an outer loop k, and thus, from an iteration of the outer loop that will already have completed (or, if $d_k < 0$ on data that has never been changed before loop k even ran).

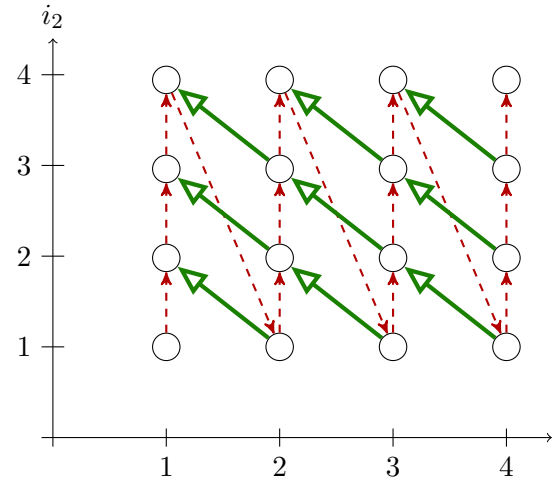
The signs in loop carrying and parallelization checks flip accordingly for loops with reverse iteration order (e.g., `ik-=4`).

If these assumptions are met, we can parallelize loop i_j, for instance using SSE3 vector processing instructions. See Chapter 4 of http://www.intel.com/Assets/PDF/manual/248966.pdf. SSE3 works on various subdivisions of 128 bit data. Automatic vectorization using single instruction multiple data (SIMD) is one very important reason why Intel compilers often achieve better performance compared to gcc.

5 Data Dependencies and Loop Optimizations

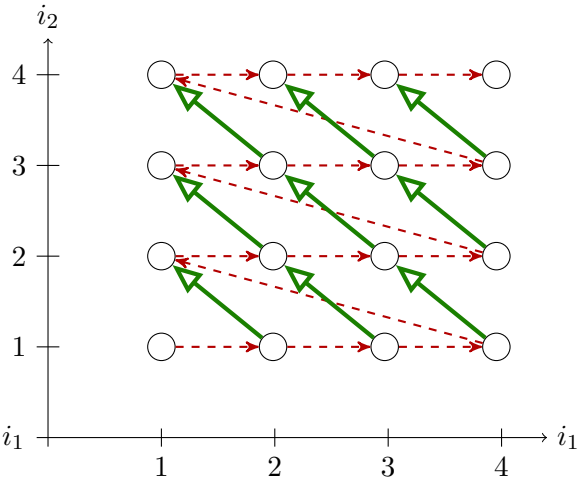
Data dependencies need to be respected in loop optimizations. The following loop uses data from the past iteration:

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



Dependency distance vector d=(1,-1)

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

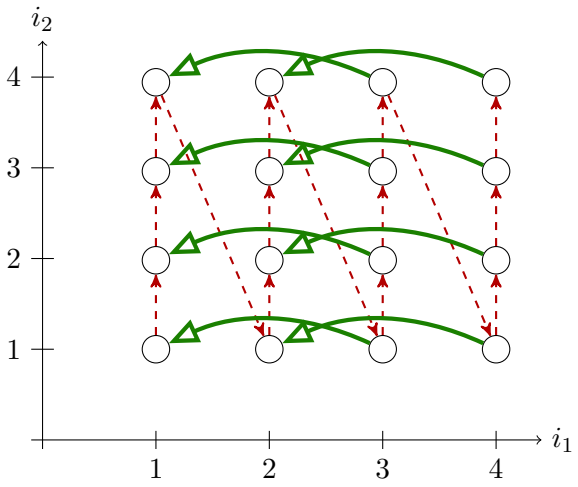


Dependency distance vector d=(-1,1)
When swapping the loops, however, we violate the data dependency, which now depends on data from a future iteration.

Note that the loop iteration order in the right example would useful for caches, because the consecutive items in cache lines are accessed subsequently, so there is a good cache hit rate. In the example on the left, elements in the cache line are accessed only very late, i.e., after walking all values of i2. For large matrices (here it's just 4) this means that the data will have been cleared from the cache again before accessing the second element in the cache line.

Data dependencies also limit parallelization and vectorization

```
for (i1=1; i1<=4; i1++)
  for (i2=1; i2<=4; i2++)
    A[i1,i2] = A[i1-2,i2]-5
```



Dependency distance vector d=(2,0)

Because of the above data dependencies, the i_2 loop can be parallelized, but the i_1 loop cannot be parallelized.

6 SIMD Vectorization and SSE / MMX

We briefly sketch how powerful vectorization can be once all proper data dependencies have been found out. 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. See newer SSE for more flexible and general vector instructions. 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 in the loop. 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 transforming data layout for SIMD usage 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.

A more fancy way to do SIMD computation with conditional branching 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]
```

compiles into

```
  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
```

Vectorization depends crucially on checking that all data dependencies even allow parallelization/vectorization. It also depends on having loop counts that fit to the size supported by the vector architecture in the CPU. The basic principle is to simply split the loop into blocks, i.e., to iterate over

blocks that are exactly of the size that fit into the vector registers. We will return to this question in lecture 25.

Some of the MMX/SSE extensions can also be used to simply have more registers available for computations.

Quiz

1. What changes with 4-way associative cache instead of a directly mapped cache? Give an example showing whether the cache miss rates are better or whether they can still be as bad as this lecture showed.

2. Why is the data dependency vector only defined for uniform array access? Is that an oversight?

3. Write down an explicit optimization (for common cases) that makes use of SSE3 instructions to optimize loops. Which side conditions do you need to check? How can you make them easier compared to the general case?

4. How do you recognize perfectly nested loops by analysis of your intermediate language?

5. How does induction variable analysis simplify for SSA?

6. Should your intermediate language representation have the for loop as a primitive? Should it have the while loop as a primitive? Should it insist on (conditional) gotos as the only way to represent looping behavior? Discuss benefits and disadvantages for various phases of your compiler.

7. Which of the dependencies (all 4 combinations of read/write after write/read) does your compiler have to worry about and for which purpose?

8. Should compiler writers try to convince chip designers to produce fully associative caches to make loop cache optimizations easier?

9. Why did the dependency distance vector flip by swapping loops. It's still the same dependency, right? Why should the vector be different after a swap?

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.