Loop Guidelines

Loops are where an application ordinarily spends the majority of its time. It is therefore useful to look in detail at how to help the compiler to produce the most efficient code.

This section describes:

- Keeping Loops Short
- Avoiding Unrolling Loops
- Avoiding Loop-Carried Dependencies
- Avoiding Loop Rotation by Hand
- Avoiding Complex Array Indexing
- Inner Loops Versus Outer Loops
- Avoiding Conditional Code in Loops
- Avoiding Placing Function Calls in Loops
- Avoiding Non-Unit Strides
- Loop Control
- Using the Restrict Qualifier

Keeping Loops Short

For best code efficiency, loops should be short. Large loop bodies are usually more complex and difficult to optimize. Large loops may also require register data to be stored in memory, which decreases code density and execution performance.

Avoiding Unrolling Loops

Do not unroll loops yourself. Not only does loop unrolling make the program harder to read but it also prevents optimization by complicating the code for the compiler.

```
// GOOD: the compiler unrolls if it helps.
void va1(const int a[], const int b[], int c[], int n) {
   int i;
   for (i = 0; i < n; ++i) {
      c[i] = b[i] + a[i];
   }
}
// BAD: harder for the compiler to optimize.
void va2(const int a[], const int b[], int c[], int n) {
   int xa, xb, xc, ya, yb, yc;
   int i;
   for (i = 0; i < n; i+=2) {
      xb = b[i]; yb = b[i+1];
```

```
      xa = a[i]; ya = a[i+1];
      xc = xa + xb; yc = ya + yb;
      c[i] = xc; c[i+1] = yc;
   }
}
```

Avoiding Loop Rotation by Hand

Do not rotate loops by hand. Programmers are often tempted to "rotate" loops in DSP code by hand, attempting to execute loads and stores from earlier or future iterations at the same time as computation from the current iteration. This technique introduces loop-carried dependencies that prevent the compiler from rearranging the code effectively. It is better to give the compiler a simpler version, and leave the rotation to the compiler.

For example,

```
// GOOD: is rotated by the compiler.
int ss(int *a, int *b, int n) {
   int sum = 0;
   int i;
   for (i = 0; i < n; i++) {
      sum += a[i] + b[i];
   }
   return sum;
}
// BAD: rotated by hand: hard for the compiler to optimize.
int ss(int *a, int *b, int n) {
   int ta, tb;
   int sum = 0;
   int i = 0;
   ta = a[i]; tb = b[i];
   for (i = 1; i < n; i++) {
      sum += ta + tb;
      ta = a[i]; tb = b[i];
   }
   sum += ta + tb;
   return sum;
}
```

Rotating the loop required adding the scalar variables `ta` and `tb` and introducing loop-carried dependencies.

Avoiding Complex Array Indexing

Other dependencies can be caused by writes to array elements. In the following loop, the optimizer cannot determine whether the load from `a` reads a value defined on a previous iteration or one that is overwritten in a subsequent iteration.

```
// BAD: has array dependency.
for (i = 0; i < n; ++i)
   a[i] = b[i] * a[c[i]];
```

The optimizer can resolve access patterns where the addresses are expressions that vary by a fixed amount on each iteration. These are known as *induction variables*.

```
// GOOD: uses induction variables.
for (i = 0; i < n; ++i)
   a[i+4] = b[i] * a[i];
```

Inner Loops Versus Outer Loops

Inner loops should iterate more than outer loops.

The optimizer focuses on improving the performance of inner loops because this is where most programs spend the majority of their time. It is considered a good trade-off for an optimization to slow down the code before and after a loop to make the loop body run faster. Therefore, try to make sure that your algorithm also spends most of its time in the inner loop; otherwise it may actually run slower after optimization. If you have nested loops where the outer loop runs many times and the inner loop runs a small number of times, try to rewrite the loops so that it is the outer loop that has fewer iterations.

Avoiding Conditional Code in Loops

If a loop contains conditional code, control-flow latencies may incur large penalties if the compiler has to generate conditional jumps within the loop. In some cases, the compiler is able to convert `if-then-else` and `?:` constructs into conditional instructions. In other cases, it can evaluate the expression entirely outside of the loop. However, for important loops, linear code should be written where possible.

There are several techniques for removing conditional code. For example, there is hardware support for `min` and `max`. The compiler usually succeeds in transforming conditional code equivalent to `min` or `max` into the single instruction. With particularly convoluted code the transformation may be missed, in which case it is better to use `min` or `max` in the source code.

The compiler can sometimes perform the loop transformation that interchanges conditional code and loop structures. Nevertheless, instead of writing

```
// BAD: loop contains conditional code.
for (i=0; i<100; i++) {
   if (mult_by_b)
      sum1 += a[i] * b[i];
   else
      sum1 += a[i] * c[i];
}
```

it is better to write

```
// GOOD: two simple loops can be optimized well.
if (mult_by_b) {
   for (i=0; i<100; i++)
      sum1 += a[i] * b[i];
} else {
   for (i=0; i<100; i++)
```

```
      sum1 += a[i] * c[i];
}
```

if this is an important loop.

Avoiding Placing Function Calls in Loops

The compiler usually is unable to generate a hardware loop if the loop contains a function call due to the expense of saving and restoring the context of a hardware loop. In addition, operations such as division, modulus, and some type coercions may implicitly call library functions. These are expensive operations which you should try to avoid in inner loops. For more details, see Data Types.

Avoiding Non-Unit Strides

If you write a loop, such as

```
// BAD: non-unit stride means division may be required.
   for (i=0; i<n; i+=3) {
         // some code
   }
```

then for the compiler to turn this into a hardware loop, it needs to work out the loop trip count. To do so, it must divide `n` by 3. The compiler may decide that this is worthwhile as it speeds up the loop, but division is an expensive operation. Try to avoid creating loop control variables with strides other than 1 or -1.

In addition, try to keep memory accesses in consecutive iterations of an inner loop contiguous. This is particularly applicable to multi-dimensional arrays.

Therefore,

```
// GOOD: memory accesses contiguous in inner loop
for (i=0; i<100; i++)
   for (j=0; j<100; j++)
      sum += a[i][j];
```

is likely to be better than

```
// BAD: loop cannot be unrolled to use wide loads.
for (i=0; i<100; i++)
   for (j=0; j<100; j++)
      sum += a[j][i];
```

as the former is more amenable to vectorization.

Loop Control

Use `int` types for loop control variables and array indices. For loop control variables and array indices, it is always better to use signed `int`s rather than any other integral type. For other integral types, the C standard requires various type promotions and standard conversions that complicate the code for the compiler optimizer. Frequently, the compiler is still able to deal with such code and create hardware loops and pointer induction variables. However, it does make it more difficult for the compiler to optimize and may occasionally result in under-optimized code.

The same advice goes for using automatic (local) variables for loop control. Use automatic variables for loop control and loop exit test. It is easy for a compiler to see that an automatic scalar whose address is not taken may be held in a register during a loop. But it is not as easy when the variable is a global or a function static.

Therefore, code such as

```
// BAD: may need to reload globvar on every iteration.
for (i=0; i<globvar; i++)
   a[i] = a[i] + 1;
```

may not create a hardware loop if the compiler cannot be sure that the write into the array `a` does not change the value of the global variable. The `globvar` must be reloaded each time around the loop before performing the exit test.

In this circumstance, the programmer can make the compiler's job easier by writing:

```
// GOOD: easily becomes a hardware loop.
int upper_bound = globvar;
for (i=0; i<upper_bound; i++)
   a[i] = a[i] + 1;
```

Using the Restrict Qualifier

The `restrict` qualifier provides one way to help the compiler resolve pointer aliasing ambiguities. Accesses from distinct restricted pointers do not interfere with each other. The loads and stores in the following loop:

```
// BAD: possible alias of arrays a and b
void copy(int *a, int *b) {
   int i;
   for (i=0; i<100; i++)
      a[i] = b[i];
}
```

may be disambiguated by writing

```
// GOOD: restrict qualifier tells compiler that memory  // accesses do not alias
void copy(int * restrict a, int * restrict b) {
   int i;
   for (i=0; i<100; i++)
      a[i] = b[i];
}
```

The `restrict` keyword is particularly useful on function parameters. but it can be used on any variable declaration. For example, the `copy` function may also be written as:

```
void copy(int *a, int *b) {
int i;
int * restrict p = a;
int * restrict q = b;

for (i=0; i<100; i++)
   *p++ = *q++;
}
```