Optimizing C++/Code optimization/Faster operations

Some elementary operations, even being conceptually as simple as others, are much faster for the processor. A clever programmer can choose the faster instructions for the job.

Though, every optimizing compiler is already able to choose the fastest instructions for the target processor, and so some techniques are useless with some compilers.

In addition, some techniques may even worsen performance on some processors.

In this section some techniques are presented that may improve performance on some compiler/processor combinations.

Contents

10 Rearrange an array of structures as several arrays

Structure fields order

Arrange the member variables of classes and structures in such a way that the most used variables are in the first 128 bytes, and then sorted from the longest object to the shortest.

If in the following structure the `msg` member is used only for error messages, while the other members are used for computations:

```
struct {
    char msg[400];
    double d;
    int i;
};
```

you can speed up the computation by replacing the structure with the following one:

```
struct {
    double d;
    int i;
    char msg[400];
};
```

On some processors, the addressing of a member is more efficient if its distance from the beginning of the structure is less than 128 bytes.

In the first example, to address the `d` and `i` fields using a pointer to the beginning of the structure, an offset of at least 400 bytes is required.

Instead, in the second example, containing the same fields in a different order, the offsets to address `d` and `i` are of few bytes, and this allows to use more compact instructions.

Now, let's assume you wrote the following structure:

```
struct {
    bool b;
    double d;
    short s;
    int i;
```

```
};
```

Because of fields alignment, it typically occupies 1 (bool) + 7 (padding) + 8 (double) + 2 (short) + 2 (padding) + 4 (int) = 24 bits (3 bytes).

The following structure is obtained from the previous one by sorting the fields from the longest to the shortest:

```
struct {
    double d;
    int i;
    short s;
    bool b;
};
```

It typically occupies 8 (double) + 4 (int) + 2 (short) + 1 (bool) + 1 (padding) = 16 bits (2 bytes). The sorting minimized the paddings (or holes) caused by the alignment requirements, and so generates a more compact structure.

Floating point to integer conversion

Exploit non-standard routines to round floating point numbers to integer numbers.

The C++ language do not provide a primitive operation to round floating point numbers. The simplest technique to convert a floating point number `x` to the nearest integer number `n` is the following statement:

```
n = int(floor(x + 0.5f));
```

Using such a technique, if `x` is exactly equidistant between two integers, `n` will be the upper integer (for example, 0.5 generates 1, 1.5 generates 2, -0.5 generates 0, and -1.5 generates -1).

Unfortunately, on some processors (in particular, the Pentium family), such expression is compiled in a very slow machine code. Some processors have specific instructions to round numbers.

In particular, the Pentium family has the instruction `fistp`, that, used as in the following code, gives much faster, albeit not exactly equivalent, code:

```
#if defined(__unix__) || defined(__GNUC__)
    // For 32-bit Linux, with Gnu/AT&T syntax
    __asm ("fldl %1 \n fistpl %0 " : "=m"(n) : "m"(x) : "memory" );
```

```
#else
    // For 32-bit Windows, with Intel/MASM syntax
    __asm fld qword ptr x;
    __asm fistp dword ptr n;
#endif
```

The above code rounds `x` to the nearest integer, but if `x` is exactly equidistant between to integers, `n` will be the nearest even integer (for example, 0.5 generates 0, 1.5 generates 2, -0.5 generates 0, and -1.5 generates -2).

If this result is tolerable or even desired, and you are allowed to use embedded assembly, then use this code. Obviously, it is not portable to other processor families.

Integer numbers bit twiddling

Twiddle the bits of integer numbers exploiting your knowledge of their representation.

A collection of hacks of this kind is here (http://www.graphics.stanford.edu/~seander/bithacks.html). Some of these tricks are actually already used by some compilers, others are useful to solve rare problems, others are useful only on some platforms.

Array cells size

Ensure that the size (resulting from the `sizeof` operator) of non-large cells of arrays or of `vector`s be a power of two, and that the size of large cells of arrays or of `vector`s be not a power of two.

The direct access to an array cell is performed by multiplying the index by the cell size, that is a constant. If the second factor of this multiplication is a power of two, such an operation is much faster, as it is performed as a bit shift. Analogously, in multidimensional arrays, all the sizes, except at most the first one, should be powers of two.

This sizing is obtained by adding unused fields to structures and unused cells to arrays. For example, if every cell is a 3-tuple of `float` objects, it is enough to add a fourth *dummy* `float` object to every cell.

Though, when accessing the cells of a multidimensional array in which the last dimension is an enough large power of two, you can drop into the *data cache contention* phenomenon (aka *data cache conflict*), that may slow down the computation by a factor of 10 or more. This phenomenon happens only when the array cells exceed a certain size, that depends on the data cache, but is about 1 to 8 KB. Therefore, in case an algorithm has to process an array whose cells have or could have as size a power of two greater or equal to 1024 bytes, first, you should detect if the data cache contention happens, e in such a case you should avoid such phenomenon.

For example, a matrix of 100 x 512 `float` objects is an array of 100 arrays of 512 `floats`. Every cell of the first-level array has a size of 512 x 4 = 2048 bytes, and therefore it is at risk of data cache contention.

To detect the contention, it is enough to add an elementary cell (a `float`) to every to every last-level array, but keeping to process the same cells than before, and measure whether the processing time decrease substantially (by at least 20%). In such a case, you have to ensure that such improvement be stabilized. For that goal, you can employ one of the following techniques:

- Add one or more unused cells at the end of every last-level array. For example, the array `double a[100][1024]` could become `double a[100][1026]`, even if the computation will process such an array up to the previous sizes.
- Keep the array sizes, but partition it in rectangular blocks, and process all the cells in one block at a time.

Prefix vs. Postfix Operators

Prefer prefix operators over postfix operators.

When dealing with primitive types, the prefix and postfix arithmetic operations are likely to have identical performance. With objects, however, postfix operators can cause the object to create a copy of itself to preserve its initial state (to be returned as a result of the operation), as well as causing the side-effect of the operation. Consider the following example:

```
class IntegerIncreaser
{
    int m_Value;

public:
    /* Postfix operator. */
    IntegerIncreaser operator++ (int) {
        IntegerIncreaser tmp (*this);

        ++m_Value;
        return tmp;
    };

    /* Prefix operator. */
    IntegerIncreaser operator++ () {
        ++m_Value;
        return *this;
    };
};
```

Because the postfix operators are required to return an unaltered version of the value being incremented (or decremented) — regardless of whether the result is actually being used — they will most likely make a copy. STL iterators (for example) are more efficient when altered with the prefix operators.

Explicit inlining

If you don't use the compiler options of whole program optimization and to allow the compiler to inline any function, try to move to the header files the functions called in bottlenecks, and declare them `inline`.

As explained in the guideline "Inlined functions" in section 3.1, every inlined function is faster, but many inlined functions slow down the whole program.

Try to declare `inline` a couple of functions at a time, as long as you get significant speed improvements (at least 10%) in a single command.

Operations with powers of two

If you have to choose an integer constant by which you have to multiply or divide often, choose a power of two.

The multiplication, division, and modulo operations between integer numbers are much faster if the second operand is a constant power of two, as in such case they are implemented as bit shifts or bit maskings.

Integer division by a constant

When you divide an integer (that is known to be positive or zero) by a constant, convert the integer to `unsigned`.

If `s` is a `signed` integer, `u` is an `unsigned` integer, and `C` is a constant integer expression (positive or negative), the operation `s / C` is slower than `u / C`, and `s % C` is slower than `u % C`. This is most significant when `C` is a power of two, but in all cases, the sign must be taken into account during division.

The conversion from `signed` to `unsigned`, however, is free of charge, as it is only a reinterpretation of the same bits. Therefore, if `s` is a `signed` integer that you *know* to be positive or zero, you can speed up its division using the following (equivalent) expressions: `(unsigned)s / C` and `(unsigned)s % C`.

Processors with reduced data bus

If the data bus of the target processor is smaller than the processor registers, if possible, use integer types not larger than the data bus for all the variables except for function parameters and for the most used local variables.

The types `int` and `unsigned int` are the most efficient, after they have been loaded in processor registers. Though, with some processor families, they could not be the most efficient type to access in memory.

For example, there are processors having 16-bit registers, but an 8-bit data bus, and other processors having 32-bit registers, but 16-bit data bus. For processors having the data bus smaller than the internal registers, usually the types `int` and `unsigned int` match the size of the registers.

For such systems, loading and storing in memory an `int` object takes a longer time than that taken by an integer not larger than the data bus.

The function arguments and the most used local variables are usually allocated in registers, and therefore do not cause memory access.

Rearrange an array of structures as several arrays

Instead of processing a single array of aggregate objects, process in parallel two or more arrays having the same length.

For example, instead of the following code:

```
const int n = 10000;
struct { double a, b, c; } s[n];
for (int i = 0; i < n; ++i) {
    s[i].a = s[i].b + s[i].c;
}
```

the following code may be faster:

```
const int n = 10000;
double a[n], b[n], c[n];
for (int i = 0; i < n; ++i) {
    a[i] = b[i] + c[i];
}
```

Using this rearrangement, "a", "b", and "c" may be processed by array processing instructions that are significantly faster than scalar instructions. This optimization may have null or adverse results on some (simpler) architectures.

Even better is to interleave the arrays:

```
const int n = 10000;
double interleaved[n * 3];
```

```
for (int i = 0; i < n; ++i) {
    const size_t idx = i * 3;
    interleaved[idx] = interleaved[idx + 1] + interleaved[idx + 2];
}
```

Remember test everything! And don't optimise prematurely.

Retrieved from "https://en.wikibooks.org/w/index.php?title=Optimizing_C%2B%2B/Code_optimization/Faster_operations&oldid=2971697"
