Loop Optimizations: taking matters into your hands

October 9, 2021Help the Compiler, Performance, Toolchain and Performance1 Reply

*We at **Johnny's Software Lab LLC** are experts in performance. If performance is in any way concern in your software project, feel free to contact us.*

We talked about the most common loop optimizations in the previous post. If you are not familiar with compiler optimizations, read it first before moving on to this post.

After you understood how the compiler optimizes your code, the two next questions are: how can you help the compiler do its job better and when does it make sense to do the optimizations manually? In this post we try to give answers to those questions, and as you will see, the answer is not simple to give.

Optimization killers

The first thing you should be aware of, the ***two biggest optimization killers are function calls and pointer aliasing***. The main reason why these are optimization killers is that, for many compiler optimizations, the compiler must assume that a certain variable is constant in the critical code it is trying to optimize. In the presence of function calls and pointer aliasing, this is much more difficult or impossible to establish, and therefore the compiler must generate slower, non-optimal machine code.

Function calls

Functions in hot loops kill the performance of your code for two reasons. The first reason is that, as far as the compiler is concerned, the function could have altered the complete state of the global memory. Take for instance the following example:

```
for (int i = 0; i < n; i++) {
   ...
   if (debug) {
       printf("The data is NaN\n");
   }
}
```

Let's say that the variable `debug` is a global variable, or a heap-allocated variable, or a member of a class. In the essence, other functions can modify its value. We, as developers, know that the variable `debug` never changes its value in this code. So, the compiler could in principle perform an optimization: loop unswitching (create two versions of the loop, one where `debug` is true and the other where `debug` is false, and dispatch to one of those versions by checking the value of `debug` outside of the loop).

However, the compiler doesn't know that, so it must assume that `printf` can modify the value of `debug`. Therefore, it won't unswitch this loop. One of the ways to solve it is to make sure that, inside the loop body, you copy the simple globally accessible variables to local variables. The functions cannot modify the values of local variables and this code is safe to optimize. Therefore, the solution would look like this:

```
bool debug_local = debug;
for (int i = 0; i < n; i++) {
    ...
    if (debug_local) {
        printf("The data is NaN\n");
    }
}
```

The compiler is safe to unswitch this loop.

The second reason why functions kill performance is the reduced ability for the compiler to optimize in the presence of a function call. Consider the following example:

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

If the compiler can inline the function `add`, it can safely perform other compiler optimizations. For example, it could vectorize the loop. But if the function is uninlinable, then the compiler must generate a scalar version of the loop and call the function `add` for each iteration of the loop.

You can improve inlining by enabling link-time optimizations to enable inlining between different compilation units.

> *When it comes to function calls in hot loops and performance, the way of the performance is to rely on automatic compiler inlining, enable link-time optimization to allow cross-module inlining or, in the worse case, to inline the function manually.*
>
> *Calls to functions, even when never executed, limit the compiler's optimization opportunities .*

Like what you are reading? Follow us on LinkedIn , Twitter or Mastodon and get notified as soon as new content becomes available.
Need help with software performance? Contact us!

Pointer aliasing

The second optimization killer is pointer aliasing. In the presence of pointer aliasing, the compiler cannot use registers to hold data, instead, it has to use slower memory. It also cannot guarantee that a certain variable is constant in a loop (because its value can be changed through another pointer). And lastly, pointer aliasing inhibits vectorization, for the reasons we explained here in more detail.

To illustrate pointer aliasing, consider the following example:

```
for (int i = 0; i < n; i++) {
   b[i] = 0;
   for (int j = 0; j < n; j++) {
      b[i] += a[i][j];
   }
}
```

For each row `i`, the compiler calculates the sum of all elements in row `i` of the matrix `a` and stores it in `b[i]`.

Assume that the compiler doesn't have pointer aliasing information about arrays `b` and `a`. Let's also assume that the matrix `a` is dynamically allocated, i.e. it is an array of pointers, each pointer pointing to another row. These are quite reasonable assumptions that can happen often.

This code has several optimization potentials. The inner loop iterates over `j`, so the value `b[i]` can be stored in a register. Additionally, the inner loop is in principle vectorizable.

But, let's assume arrays `a` and `b` were initialized in the following manner:

```
double** a = new double*[n];
double* b = new double[n];
for (int i = 0; i < n; i++) {
    a[i] = b;
}
```

In the matrix `a`, all the row pointers point to the same block of memory. Additionally, rows of `a` and pointer `b` alias each other. If you write a value 5 to b[5], this value will appear at the location `a[3][5]`.

In this case, the compiler cannot use a register to hold `b[i]`. Also, it cannot vectorize the loop because of the dependencies. The result is poorly generated assembly.

You will say that this never happens in real codebases, but the compiler must assume the worst-case scenario. The compiler will not throw a warning, you will need to look at the compiler's optimization report to see what happened. We will talk about the compiler's optimization report in the next post.

If you are sure that two pointers do not alias each other, then you can use the `__restrict__` keyword to tell the compiler that pointers are independent of one another. Additionally, you can manually replace the writes to the same element of the array with a register, like this:

```
for (int i = 0; i < n; i++) {
   double sum = 0;
   double* __restrict__ a_row = a[i];
   for (int j = 0; j < n; j++) {
      sum += a_row[j];
   }
   b[i] = sum;
}
```

By using the `__restrict__` keyword for the row of matrix `a` (line 3), and introducing a temporary scalar variable to hold the intermediate result, the compiler is now free to better optimize the loop.

A few additional notes about pointer aliasing

Pointer aliasing is one of the most complex analyses the compiler does, and many compiler optimizations can only be done when the compiler can guarantee there is no pointer aliasing. That being said, it is of great importance to simplify the analysis for the compiler by using local variables instead of globals, using the `__restrict__` keyword or telling the compiler to ignore loop carried dependencies using compiler pragmas (e.g. `#pragma ivdep`).

When analyzing your code, with respect to pointer aliasing, the compiler distinguishes between three types of data (by data we mean scalar variables or arrays):

1. The compiler is sure that the data is not aliased by other pointers, so it is safe to do compiler optimizations on it.
2. The compiler is sure that the data is aliased by other pointers, so it must assume that its value can change, omit certain compiler optimizations and keep the data in memory, not register.
3. The compiler cannot guarantee that the data is not aliased by pointers.

In the case of (1) and (2), the solution is clear. In the case of (3), the compiler has an option to generate two versions of the code: an optimized version when there is no pointer aliasing, and a non-optimized version for the case of pointer aliasing. Then, it emits runtime checks for pointer aliasing and accordingly selects the faster or the slower path.

Note, however, that runtime aliasing analysis is limited in several ways: it works only on scalars and arrays for which the compiler can deduce the length. Since each pointer needs to be checked with each other pointer, the computation needed to perform the analysis can quickly become unmanageable. That’s why you should not rely too much on it.

For the hot loop, inspecting the compiler optimization report together with copying global values to local, using the `__restrict__` keyword, telling the compiler to ignore vector dependencies on a hot loop using a pragma (e.g. `#pragma ivdep`) will allow compiler optimizations.

From the readibility perspective, copying globals to locals and using locals instead yields a more readable code. The `__restrict__` keyword is not known to many developers, and if used incorrectly, will result in incorrect program behavior. The same applies for compiler pragmas.

Taking matter into your own hands

Before taking matters into your hand, I must make a warning. In my experience, code readability and code maintainability are higher in importance than speed. Optimizing non-critical parts of code never brings a relevant increase in speed. So, all the changes you plan to do to the code should be focused only on critical loops.

Loop Invariant Code Motion and Loop Unswitching

You should try to let the compiler do the loop invariant code motion and loop unswitching as much as possible. However, under some circumstances, you want to do them manually:

- **The compiler cannot establish that the parameter is a constant**: same rules apply here as with pointer aliasing. Either copy the loop invariant condition to a temporary local variable, or convert it to compile time constant using macros or C++ templates.
- **There are too many parameters for the compiler to unswitch the loop**: loop unswitching doubles the code size; with too many parameters at one point it will stop unswitching. Manual unswitching using C++ templates or macros will force unswitching.

More information about manual loop unswitching can be found in the post about flexibility and performance.

Removing Iterator Variable Dependent Computations

If there is a condition in the loop, and it doesn't depend on the data but only on the iterator variable, we call it the iterator variable dependent condition. The compilers typically do not optimize these conditions away, so it is almost always useful to get rid of them, either through loop peeling or loop unrolling.

Loop Unrolling

In my experience, you should almost never do loop unrolling by hand. Manual loop unrolling complicates the analysis for the compiler, and the resulting code can be slower. You can, however, use compiler's pragmas to force loop unrolling on the compiler (e.g. LLVM offers `pragma clang loop unroll`).

UPDATE: Some readers objected to this observation, and I feel there should be an update to it. Often developers unroll loops a few times, and then rearrange the statements in the loop, similar to loop pipelining. What happens in the background is that the instructions corresponding to those statements move together with the statements. If you are lucky enough, a CPU unit that was the bottleneck earlier won't be a bottleneck anymore and your loop will run faster.

However, this kind of "performance tuning" is not necessarily portable, not between different compilers, nor between different hardware architectures on the same compiler, nor between different versions of the same compiler. Compilers are free to schedule instructions anyway they please as long as the results remain the same. The performance obtained using this approach tend to be small (e.g. 20% decrease in program runtime) and not portable. The code is more complex.

That being said, sometimes you need that extra 20% and it makes sense to "hack" it in this way. Sometimes this saves money, time or power. And maybe in that exact case that is more important than maintainability and portability.

Loop Pipelining

As far as loop pipelining is concerned, it is best left to the compiler to this optimization, and only for those architectures that would benefit from it.

There is one exception to this rule: if your loop is accessing data in an unpredictable pattern, you can expect a large amount of data cache misses which can tremendously slow down your code. If this is the case, explicit software prefetching in a combination with loop pipelining can help mitigate some of the problems. The technique is quite involved, but can be worth the effort. You can find more about it here.

Vectorization

The only way to vectorize the code manually is to use vectorization pragmas, such as `#pramga omp simd` (portable, however, you need to provide -fopenmp or -fopenmp-simd switch to the compiler), or a compiler-specific one, like LLVM's `#pragma clang loop vectorize(enable)`. Nevertheless, I am against using vectorization pragmas. Here is why.

In case the compiler didn't vectorize the loop, there is a specific reason. It can be:

1. The compiler didn't know how to vectorize the loop: forcing vectorization won't make a difference, except creating a compiler warning.
2. There were loop carried dependencies: forcing vectorization will create wrong results, so this is a mistake.
3. The cost model predicted that the vectorization doesn't pay off: if this is the case, forcing vectorization will probably result in a slowdown, not speed up.
4. The results will not be precise according to IEEE 754 standard: enable compiler flag that allow relaxed IEEE 754 semantics, like `-ffast-math`, `-Ofast`, `-fassociative-math`, etc. When you do this, the compiler will automatically vectorize the loop without pragma.
5. The compiler couldn't guarantee there is no pointer aliasing: if this is the case, tips we mentioned in the section about pointer aliasing will help the compiler perform the pointer aliasing analysis successfully, so it can automatically vectorize the loop if the cost model predicts speedups.
6. The loop was too complex for the compiler to vectorize. Compiler's vectorization pass works by recognizing patterns. When the loop is too complex, it fails, even if the loop is vectorizable. In this case you can vectorize the loop manually using vectorization intrinsics or some other.

True, in some rare cases, forcing vectorization might bring performance benefits (e.g. if you don't want to relax the precision for IEEE 754 floating-point mathematics, except for one particular place), but these are rarely seen in practice.

Vectorization is a huge topic and requires a lot of place to cover completely. If your hot loop is not vectorized, a useful resource on how to do it is Sergey Slotin's book.

Loop Interchange

The compilers rarely do loop interchange, and it is a very valuable transformation to speed up codes that work with matrices and images. It should be definitely done manually for the loops on the hot code because it has the capacity to increase its speed several times.

Loop Distribution

Loop distribution is a somewhat controversial optimization. If the loop is unvectorizable, the optimization can split the loop into vectorizable and unvectorizable parts. Vectorization of one part should bring an increase in speed. However, in practice, there is a problem with this approach. Consider the following example:

```
for (int i = 0; i < n; i++) {
    double val = (a[i] > 0) ? a[i] : 0;
    b[i] = sqrt(a[i]);
}
```

Operation `sqrt` is an expensive operation and this loop would benefit from vectorization. Some compilers are better at vectorization than others. Now, we can perform manual loop distribution like this:

```
for (int i = 0; i < n; i++) {
    val[i] = (a[i] > 0) ? a[i] : 0;
}
for (int i = 0; i < n; i++) {
    b[i] = sqrt(val[i]);
}
```

Let’s say that compiler A didn’t vectorize the original loop. After the distribution, compiler A vectorized the second loop, and the overall impact on the speed is positive.

However, if the compiler B vectorized the original loop, after the distribution, the impact of distribution on performance will be negative.

Although loop distribution can have a positive impact on speed, it is important to measure the performance impact on all the compilers your project uses.

Loop Fusion

Loop Fusion generally has a positive impact on performance, with one exception: if one (or both) of the original loops were vectorized by the compiler, and after the fusion they stop being vectorized, the impact can be negative. Same as with loop distribution, it is important to check the performance with various compilers.

Conclusion

There are a few concerns when it comes to taking matters in your own hands with regards to compiler optimizations. The first is code readability and maintainability. Code readability is very important almost all of the time since it decreases maintenance costs: well-written code is easier for another person to pick up, it has fewer chances of having bugs and in the long run it is easier to extend.

Manual code optimizations can result in code that is messy and more difficult to maintain. And knowing that performance optimizations only matter for the hot code, you should do them only on hot loops and only after thorough profiling.

The second concern is performance portability: there is no guarantee that speedups achieved with one compiler will reproduce with another compiler. So it is important to taker that into account too.

Nonetheless, if performance optimizations can result that the product can be shipped, its price can go down, or it uses less power, it is definitely worth paying attention to the low-hanging fruit of forgotten compiler optimizations.

In the next post, we will talk about how to use the compiler optimization report with CLANG, using a very nice graphical UI called `opt-viewer.py`.

> *Like what you are reading? Follow us on LinkedIn , Twitter or Mastodon and get notified as soon as new content becomes available.*
> *Need help with software performance? Contact us!*

Further Read

The joys and perils of C and C++ aliasing, Part 1

The joys and perils of C and C++ aliasing, Part 2