Loop Optimizations: how does the compiler do it?

*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.*

Modern compilers are indispensable when it comes to the performance of modern software. If you have been developing even for a few weeks, you know that already. You know that the difference in speed between a Debug build, with all optimizations disabled, and a Release build, with optimizations enabled, can be an order of magnitude. But how do the compilers achieve this wonder? What is the catch?

In this post we are not going to give a full explanation of each optimization the compiler performs on loops. Instead, we are going to give a brief summary of the most common optimizations the compiler does on loops and their effect on performance. In the follow-up post we talk about `opt-viewer`, a LLVM tool that allows you to simply inspect the compiler optimizations and in the final post we talk about helping the compiler do a better job at optimizing.

The perfect loop

Before going into the details, let's first what is are the goals of the compiler? What kind of loop should it produce? The ideal loop should have the following properties:

- Overall, the loop should perform only the essential work. This is achieved by removing all redundant or unnecessary computations outside of the loop.
- It should use the fastest available instructions to achieve the goal.
- The loop should strive to use SIMD instructions as much as possible. We wrote a lot about SIMD instructions here. In the essense, they allow the CPU to process more than one piece of data in one instruction. For instance, the CPU can load 4 doubles from the memory, perform 4 increments and then perform 4 stores of the modified value back to the memory.
- The instructions inside the loop should be generated in such a way to use the different hardware units of the CPU in a balanced way. If a certain unit of the CPU is oversubscribed, like divisor or the load unit,

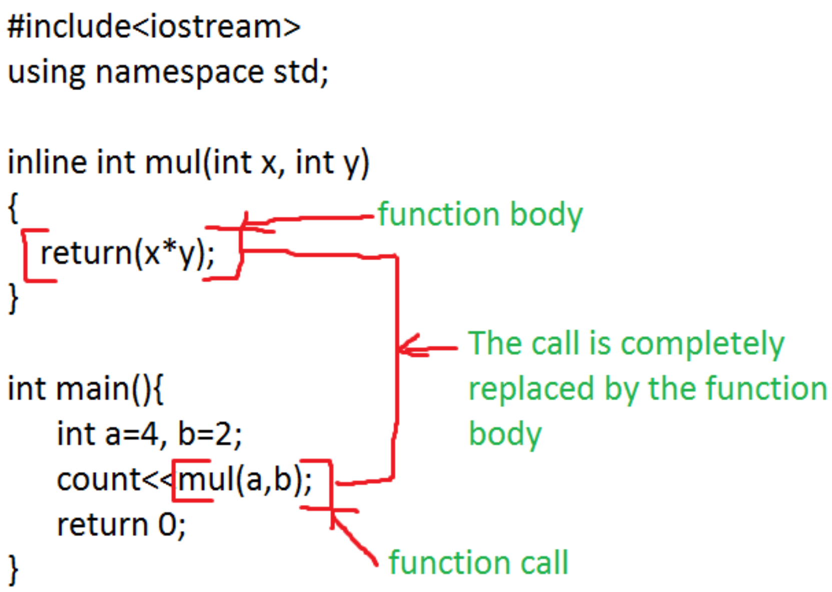
then the bottleneck there will make the whole loop run slower than it needs to.
- The memory access pattern inside the loop should be as good as possible. A good memory access pattern translates to better performance. Ideally, the loop should access data stored in an array, going from 0 to N with an increment of 1.

If the compiler manages to generate the loop with such properties, this will guarantee the peak performance possible for the loop.

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!

Before Beginning – Inlining

Before the compiler starts working hard on optimizing your loop, it performs inlining. Inlining means replacing the call to the function inside the loop with the code from the body of the function. As they say, an image is worth a thousand words:



```
#include<iostream>
using namespace std;

inline int mul(int x, int y)
{
  return(x*y);
}

int main(){
    int a=4, b=2;
    count<<mul(a,b);
    return 0;
}
```

Inlining

Inlining itself is a small compiler optimization because it removes some overhead of calling another function. But the main benefit of inlining comes from enabling other optimizations. In the above case, after the function `mul` is inlined, the compiler can calculate the value of `a*b` and replace it with a constant `8`.

We wrote about inlining extensively in the past, but here is the short summary: the compiler will automatically inline short functions based on some criterion if they are in the same file. If not, you can enable inlining across different files by putting the functions you wish to inline in a common header. Alternatively, you can enable link-time optimizations.

Basic Optimizations

First, we focus our attention on the most basic optimizations, the ones that should happen every time when possible.

Keeping variables in registers

The fastest type of memory storage in the CPU are registers, second to it is the memory. Even when the piece of data is strictly in the fastest level of the data cache, accessing it is still slower than when accessing it in a register.

Compilers have a component called *register allocator* which allocates registers to hold values of certain variables. The number of registers in the CPU is limited, and depending on the architecture it is somewhere between 8 and 32. The compiler decides, based on some metrics, which variables to keep in registers and which variables to keep in memory.

Normally, a simple scalar value[1] can be stored in a register. The program can store in a register an integer, a double, a pointer, or a simple member of a class. The compiler cannot store the value of a full-blown class in a register.

There are two things that can prevent the compiler from storing the value in the register:

- **Too many variables**: if inside your loop you have too many used variables, the compiler cannot store all of them in registers. Since many instructions work efficiently only on registers, the compiler will need to remove the currently unused values to memory in the process called *register spilling*. Later, when the values are needed again, the compiler will

need to reload them back from the memory to the registers. Spilling does no useful work for the program, and the compilers try to spill as least as possible.
- **Pointer aliasing**: if a scalar value is pointed by a pointer, this means it can be modified in two ways: directly or through a pointer. The compiler cannot store scalar value in a register, because modification done to it through a pointer would be lost. Therefore, every modification to the variable must be done in a load-modify-store group of operations.

The problem of register spilling due to too many variables can be solved by splitting the loop into two. Intel's documentation gives an example of such a loop and proposes a way to solve it. Before doing that, make sure there actually is a register spilling problem by observing the compiler optimization report.

The problem of pointer aliasing is very well illustrated in this post. Both C and C++ have strict-aliasing semantics, which means if you have pointers and scalars of the same type, like `int* p` and `int i`, unless it can guarantee that `p` doesn't point to `i`, it must assume that `p` can point to `i` and therefore `i` cannot be held in a register.

The solution to this problem is to let the compiler know that the scalar is safe to hold in a register. For global variables, static variables, and class data members it can be done by keeping a copy of the variable in a local automatic variable. The compiler can easily figure out that there are no pointers point to it since the variable is local and therefore can allocate a register for the variable.

Removing redundant computation

The compiler’s goal is to remove as much as redundant computation from the loop. There are two types of redundant computation.

Unneded computation

Some computation is never needed. Take the following example:

Plain text
Copy to clipboard
Open code in new window
EnlighterJS 3 Syntax Highlighter

```
void add(int* a, int* b) {
(*a)++;
if (b) (*b)++;
}
for (int i = 0; i < n; i++) {
add(&a[i], nullptr);
}
```

```
void add(int* a, int* b) { (*a)++; if (b) (*b)++; }
for (int i = 0; i < n; i++) { add(&a[i], nullptr); }
```

```
void add(int* a, int* b) {
    (*a)++;
    if (b) (*b)++;
}

for (int i = 0; i < n; i++) {
    add(&a[i], nullptr);
}
```

After the compiler inlines the function add, the condition `if(b)` on line 3 is always false and line 3 can be completely removed. What remains is a simple loop with `a++` in its body.

The compiler will omit dead code (*dead code elimination*) whenever it can because it both decreases the code size and makes the loop faster. Sometimes, however, it fails to do so because it cannot figure out that the code is never executed. Looking at the compiler optimization report can help.

Loop invariant computation

Loop invariant computation is needed in the loop, but its value doesn’t need to be calculated repeatedly in every iteration of the loop, since it never changes. Consider the following example

Plain text
Copy to clipboard
Open code in new window
EnlighterJS 3 Syntax Highlighter

```
for (int i = 0; i < n; i++) {
switch (operation) {
case ADD: a[i]+= x * x; break;
case SUB: a[i]-= x * x; break;
}
}
```

for (int i = 0; i < n; i++) { switch (operation) { case ADD: a[i]+= x * x; break; case SUB: a[i]-= x * x; break; } }

```
for (int i = 0; i < n; i++) {
    switch (operation) {
        case ADD: a[i]+= x * x; break;
        case SUB: a[i]-= x * x; break;
    }
}
```

The variables `operation` and `x` are loop invariant, i.e. they don’t change the value while the loop is running. A smart compiler can then calculate the value of the expression `x*x` outside of the loop and reuse it everywhere in the loop. This optimization is called *loop invariant code motion*. Compilers do this optimization every time they can, under the condition they can establish that the expression is loop invariant.

In the case of variable `operation`, the situation is a bit more complex. The variable is loop invariant, but it determines the control flow inside the loop. The compiler can do the following: create a separate version of the loop for each possible value of the variable `operation`. The transformation is called *loop unswitching*, because there is a different version of the loop for each value of the condition. Here is the loop from the previous example after loop unswitching on variable `operation` and loop invariant code motion on expression `x*x`.

Plain text
Copy to clipboard
Open code in new window
EnlighterJS 3 Syntax Highlighter

```
auto x_2 = x * x;
if (operation == ADD) {
for (int i = 0; i < n; i++) {
a[i] += x_2;
}
} else if (operation == SUB) {
for (int i = 0; i < n; i++) {
a[i] -= x_2;
}
}
```

auto x_2 = x * x; if (operation == ADD) { for (int i = 0; i < n; i++) { a[i] += x_2; } } else if (operation == SUB) { for (int i = 0; i < n; i++) { a[i] -= x_2; } }

```
auto x_2 = x * x;
if (operation == ADD) {
    for (int i = 0; i < n; i++) {
        a[i] += x_2;
    }
} else if (operation == SUB) {
    for (int i = 0; i < n; i++) {
        a[i] -= x_2;
    }
}
```

As opposed to loop invariant code motion, the compilers are more careful with loop unswitching. The size of the code can grow exponentially with the number of unswitched variables, therefore after some threshold, they tend to omit it. If your hot loop could benefit from loop unswitching, you should definitely check if the compiler does it, and if not, do it manually.

The main challenge in loop invariant optimizations is detecting the loop invariance. Often the compiler cannot guarantee that a certain variable is loop invariant, due to pointer aliasing or too complex code. In that case it has to play safe and generate the non-optimal code.

Iterator variable dependent computation

Iterator variable dependent computations are computations that depend on the value of the iterator variable, and not on the data. Consider the following example:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

```
for (int i = 0; i < n; i++) {
auto min_val = a[i];
if (i != 0) {
min_val = std::min(a[i - 1], min_val);
}
if (i ≠ (n - 1)) {
min_val = std::min(a[i + 1], min_val);
}
b[i] = min_val;
}
```

for (int i = 0; i < n; i++) { auto min_val = a[i]; if (i ≠ 0) { min_val = std::min(a[i - 1], min_val); } if (i ≠ (n - 1)) { min_val = std::min(a[i + 1], min_val); } b[i] = min_val; }

```
for (int i = 0; i < n; i++) {
    auto min_val = a[i];
    if (i ≠ 0) {
        min_val = std::min(a[i - 1], min_val);
    }
    if (i ≠ (n - 1)) {
        min_val = std::min(a[i + 1], min_val);
    }
    b[i] = min_val;
}
```

The conditions on line 3 and line 6 depend on the value of the iterator, they do not depend on the data. As such, we remove them from the loop by moving them outside of the loop and giving them special treatment.

Plain text
Copy to clipboard
Open code in new window
EnlighterJS 3 Syntax Highlighter

```
b[0] = std::min(a[0], a[1]);
for (int i = 1; i < n - 1; i++) {
auto min_val = a[i];
min_val = std::min(a[i - 1], min_val);
min_val = std::min(a[i + 1], min_val);
b[i] = min_val;
}
b[n - 1] = std::min(a[n - 2], a[n - 1]);
```

b[0] = std::min(a[0], a[1]); for (int i = 1; i < n - 1; i++) { auto min_val = a[i]; min_val = std::min(a[i

```
- 1], min_val); min_val = std::min(a[i + 1],
min_val); b[i] = min_val; } b[n - 1] = std::min(a[n -
2], a[n - 1]);
```

```
b[0] = std::min(a[0], a[1]);

for (int i = 1; i < n - 1; i++) {
    auto min_val = a[i];
    min_val = std::min(a[i - 1], min_val);
    min_val = std::min(a[i + 1], min_val);
    b[i] = min_val;
}

b[n - 1] = std::min(a[n - 2], a[n - 1]);
```

Compilers rarely or ever do such optimizations. A common misconception is that on modern CPUs, conditions that are trivial to predict are essentially free. This is true, but conditions in the loop body inhibit other useful compiler optimizations, and almost always you will see some speed improvements if you do this optimization manually.

Using the cheaper instructions

Compilers should use the cheapest instruction whenever possible. We wrote about replacing expensive instructions with cheaper ones a while ago, but in the context of the loops, there are a few more things to say.

One of the very common optimization techniques is called *induction variables*. Consider the following example:

Plain text
Copy to clipboard
Open code in new window
EnlighterJS 3 Syntax Highlighter

```
for (int i = 0; i < n; i++) {
a[i] = i * 3;
}
for (int i = 0; i < n; i++) { a[i] = i * 3; }
for (int i = 0; i < n; i++) {
    a[i] = i * 3;
}
```

The expression `i * 3` on line 2 is a linear function of iterator variable `i`. So, instead of performing the multiplication in each iteration, the compiler can keep the current value, and simply add offset from the previous value using addition. Like this:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

```
tmp = 0;
for (int i = 0, int tmp = 0; i < n; i++) {
a[i] = tmp;
tmp += 3;
}
tmp = 0; for (int i = 0, int tmp = 0; i < n; i++) {
a[i] = tmp; tmp += 3; }
tmp = 0;

for (int i = 0, int tmp = 0; i < n; i++) {
    a[i] = tmp;
    tmp += 3;
}
```

This technique is especially useful in the context of access to array elements. For example, we have a class `MyClass` and `sizeof(MyClas)` is 24. Here is the loop that iterates over the instances of the class:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

```
class MyClass {
double a;
double b;
double c;
};
for (int i = 0; i < n; i++) {
a[i].b += 1.0;
}
```

class MyClass { double a; double b; double c; }; for (int i = 0; i < n; i++) { a[i].b += 1.0; }

```
class MyClass {
    double a;
    double b;
    double c;
};

for (int i = 0; i < n; i++) {
    a[i].b += 1.0;
}
```

The member `b` is at offset 8. So, to calculate the address of `a[i].b`, the compiler translates it to `(a + i * sizeof(MyClass) + 8`. But, for this particular loop, the compiler can take a shortcut and instead of using this formula, it knows that the address of the first element is `a + 8`, and the address of the next element is the address of the current element plus `sizeof(MyClass)`. This reduces the operation of address calculation from multiplication and two additions to only one addition.

Loop unrolling and related optimizations

After doing the basic optimizations, the compiler moves to the next phases of optimizations. The next phase is loop unrolling. Loop unrolling is best described using an example. Consider the following code, loop that calculates `a[i] = b[i/2]` written in pseudo-assembler:

Plain text
Copy to clipboard
Open code in new window
EnlighterJS 3 Syntax Highlighter

```
for (int i = 0; i < n; i++) {
index = i / 2;
b_val = load(b + index);
store(a + i, b_val);
}
```

for (int i = 0; i < n; i++) { index = i / 2; b_val = load(b + index); store(a + i, b_val); }

```
for (int i = 0; i < n; i++) {
    index = i / 2;
    b_val = load(b + index);
    store(a + i, b_val);
}
```

When the loop is this small, the overhead of the loop itself can be as large as the work done inside the loop. In this particular case, the overhead of assignment `a[i] = b[i/2]` can be equal to evaluating the exit condition `i < n` and performing the jump to the beginning of the loop.

With loop unrolling, the idea is to replicate the body of the loop a few times. With each replication, the body of the loop becomes longer so the loop overhead decreases. Here is the above loop unrolled four times (with the factor of four):

Plain text
Copy to clipboard
Open code in new window
EnlighterJS 3 Syntax Highlighter

```
for (int i = 0; i < n; ) {
index = i / 2;
b_val = load(b + index);
store(a + i, b_val);
i++;
index = i / 2;
b_val = load(b + index);
store(a + i, b_val);
i++;
index = i / 2;
b_val = load(b + index);
store(a + i, b_val);
i++;
index = i / 2;
b_val = load(b + index);
store(a + i, b_val);
i++;
}
for (int i = 0; i < n; ) { index = i / 2; b_val =
load(b + index); store(a + i, b_val); i++; index = i
/ 2; b_val = load(b + index); store(a + i, b_val);
i++; index = i / 2; b_val = load(b + index); store(a
+ i, b_val); i++; index = i / 2; b_val = load(b +
index); store(a + i, b_val); i++; }
```

```
for (int i = 0; i < n; ) {
    index = i / 2;
    b_val = load(b + index);
    store(a + i, b_val);
    i++;

    index = i / 2;
    b_val = load(b + index);
    store(a + i, b_val);
    i++;

    index = i / 2;
    b_val = load(b + index);
    store(a + i, b_val);
    i++;

    index = i / 2;
    b_val = load(b + index);
    store(a + i, b_val);
    i++;
}
```

There are two benefits of unrolling: it decreases the overhead of the loop, and it also allows for some additional optimizations. For the above case, the compiler knows that to values of `i / 2` and `(i + 1) / 2` are the same if `i` is even, so it can remove some of the unneeded loads:

Plain text
Copy to clipboard
Open code in new window
EnlighterJS 3 Syntax Highlighter

```
for (int i = 0; i < n; ) {
index = i / 2;
b_val = load(b + index);
store(a + i, b_val);
i++;
store(a + i, b_val);
i++;
index = i / 2;
b_val = load(b + index);
store(a + i, b_val);
i++;
store(a + i, b_val);
i++;
}
```

for (int i = 0; i < n;) { index = i / 2; b_val = load(b + index); store(a + i, b_val); i++; store(a + i, b_val); i++; index = i / 2; b_val = load(b + index); store(a + i, b_val); i++; store(a + i, b_val); i++; }

```
for (int i = 0; i < n; ) {
    index = i / 2;
    b_val = load(b + index);
    store(a + i, b_val);
    i++;
    store(a + i, b_val);
    i++;

    index = i / 2;
    b_val = load(b + index);
    store(a + i, b_val);
    i++;

    store(a + i, b_val);
    i++;
}
```

After the optimization which was made possible by loop unrolling, the loop has only two loads from memory compared to the original which had four loads.

Compilers typically schedule instructions on a basic block level.[2]. Loop unrolling increases the size of the basic block, and with it, it increases the opportunities for better instruction scheduling. With the increased basic block size, the compiler can schedule instructions better, with the goal of increasing the available instruction-level parallelism, using the CPU resources in a more balanced way, etc.

There are a few drawbacks of loop unrolling:

- **Loop unrolling increases pressure on the memory subsystem, especially the instruction cache and instruction decoding units of the CPU**. The instruction cache is limited in capacity, and with large loops you can get into a problem of cache trashing, where the current instructions evict the instructions that were already used but will still be used later. This slows down the program execution so the compilers are conservative when it comes to loop unrolling.

- **The loop overhead is almost negligible on modern out-of-order CPUs, even for very small loops**. So, aggressive loop unrolling as a compiler optimization technique belongs to the past.

A few additional notes. The compilers tend to unroll small loops with fixed iteration counts completely. So instead of having a loop that goes from 0 to 5, you will have 5 times repeated the body of the loop and no loop.

In the earlier times, when the compiler optimizations were still in development, one could get some speed improvements by manually unrolling a loop. Don't do that now! Manual loop unrolling hinders other compiler optimization; manually unrolled loops are more difficult for the compiler to analyze and the resulting code can actually be slower. You can control loop unrolling factor using compiler pragmas, for instance in CLANG, specifying `pragma clang loop unroll factor(2)` will unroll the following loop by a factor of two.

Loop Pipelining (or Software Pipelining)

One of the things that limit the computational speed of CPUs are *instructions dependencies*. If there are two instructions, A and B, and the source operand of B is a result of A, we say that there is a dependency between B and A.

Here is a very simple example of a loop calculating `c[i] = a[i] + b[i]`, written in a pseudo assembler:
Plain text
Copy to clipboard
Open code in new window

EnlighterJS 3 Syntax Highlighter

```
for (int i = 0; i < n; i++) {
val_a = load(a + i);
val_b = load(b + i);
val_c = add(val_a, val_b);
store(val_c, c + i);
}
```

for (int i = 0; i < n; i++) { val_a = load(a + i); val_b = load(b + i); val_c = add(val_a, val_b); store(val_c, c + i); }

```
for (int i = 0; i < n; i++) {
   val_a = load(a + i);
   val_b = load(b + i);
   val_c = add(val_a, val_b);
   store(val_c, c + i);
}
```

The loop consists of four instructions, two loads (lines 2 and 3), an addition (line 4) and a store (line 5). Two loads do not depend on one another, however, addition requires that both loads have finished and the store requires that the addition has finished.

This *chain of dependency* prevents the CPU from achieving the maximum throughput of instructions. For example, the CPU has to wait for the load operations to complete before moving on to add operations. Even though each iteration can be done independently, the dependency inside a single iteration limits the available instruction-level parallelism. The compilers employ a technique called *loop pipelining* to break the dependency chain.

The idea behind loop pipelining is to split the work done inside a single iteration into phases, and then execute phase 1 from iteration 0, phase 2 from iteration 1, etc. This explanation is probably too abstract, but we'll explain it using the previous loop.

Let's perform software pipelining on the loop from the previous example. We split the loop body into three phases: loading of inputs phase, addition phase and storing the result phase. In the single loop body, we perform load for iteration `i + 2`, addition for iteration `i+1` and storage for iteration `i`. The code looks like this:

Plain text
Copy to clipboard
Open code in new window
EnlighterJS 3 Syntax Highlighter

```
val_a = load(a + 0);
val_b = load(b + 0);
val_c = add(val_a, val_b);
val_a = load(a + 1);
val_b = load(b + 1);
for (int i = 0; i < n - 2; i++) {
store(val_c, c + i);
val_c = add(val_a, val_b);
val_a = load(a + i + 2);
val_b = load(b + i + 2);
}
store(val_c, n - 2);
val_c = add(val_a, val_b);
store(val_c, n - 1);
```

val_a = load(a + 0); val_b = load(b + 0); val_c = add(val_a, val_b); val_a = load(a + 1); val_b = load(b + 1); for (int i = 0; i < n - 2; i++) { store(val_c, c + i); val_c = add(val_a, val_b); val_a = load(a + i + 2); val_b = load(b + i + 2); } store(val_c, n - 2); val_c = add(val_a, val_b); store(val_c, n - 1);

```
val_a = load(a + 0);
val_b = load(b + 0);
val_c = add(val_a, val_b);
```

```
val_a = load(a + 1);
val_b = load(b + 1);

for (int i = 0; i < n - 2; i++) {
    store(val_c, c + i);
    val_c = add(val_a, val_b);
    val_a = load(a + i + 2);
    val_b = load(b + i + 2);
}

store(val_c, n - 2);

val_c = add(val_a, val_b);
store(val_c, n - 1);
```

Let’s first focus on the loop body (lines 9-12). On line 9, we store the result for iteration `i`. On line 10, we calculate the addition for iteration `i + 1` and on lines 11 and 12 we load the operands for iteration `i + 2`. Inside a single loop iteration, `load`, `add` and `store` do not depend on each other. The CPU can execute them all in a single cycle.

The pipelined loop also has a prologue (lines 1-6) and epilogue (lines 15-18) that complete all the instructions needed for complete processing. Loop pipelining in this example implies that the loop minimum trip count is two.

Modern out-of-order CPUs (desktop and server systems) do not benefit a lot from loop pipelining because they can execute other instructions that come later in the instructions stream. However, in-order CPUs (simple, low-powered CPUs used in mobile phones, IoT, etc) often benefit greatly from this kind of optimization.

Vectorization and related optimizations

After doing the optimizations we mentioned until now, the compiler should have a code that is sleek, without unneeded computations and only with essential dependencies. However, in many circumstances, one can extract a lot more speed through *vectorization*.

We wrote extensively about vectorization in the earlier post about software parallelization and SIMD , which we suggest to read if you are interested in the topic. The basic idea behind vectorization is that there are special *vector registers* in the CPU that can hold more than one value of the same type. For example, a vector register of the AVX x86-64 instruction set architecture is 256 bits wide and can hold 4 doubles, or 8 floats, or 16 shorts or 32 chars. The number of values a register can hold is also called *a number of lanes*.

The CPU also has special instructions that work with all the values in the vector registers at once. For example, the CPU can load 4 doubles in one instruction, add together two pairs of 4 doubles or write 4 resulting doubles back to memory.

For example, a simple loop doing `A[i] = B[i] + C[i]` on doubles will result in the following pseudo-assembler:

Plain text

Copy to clipboard

Open code in new window

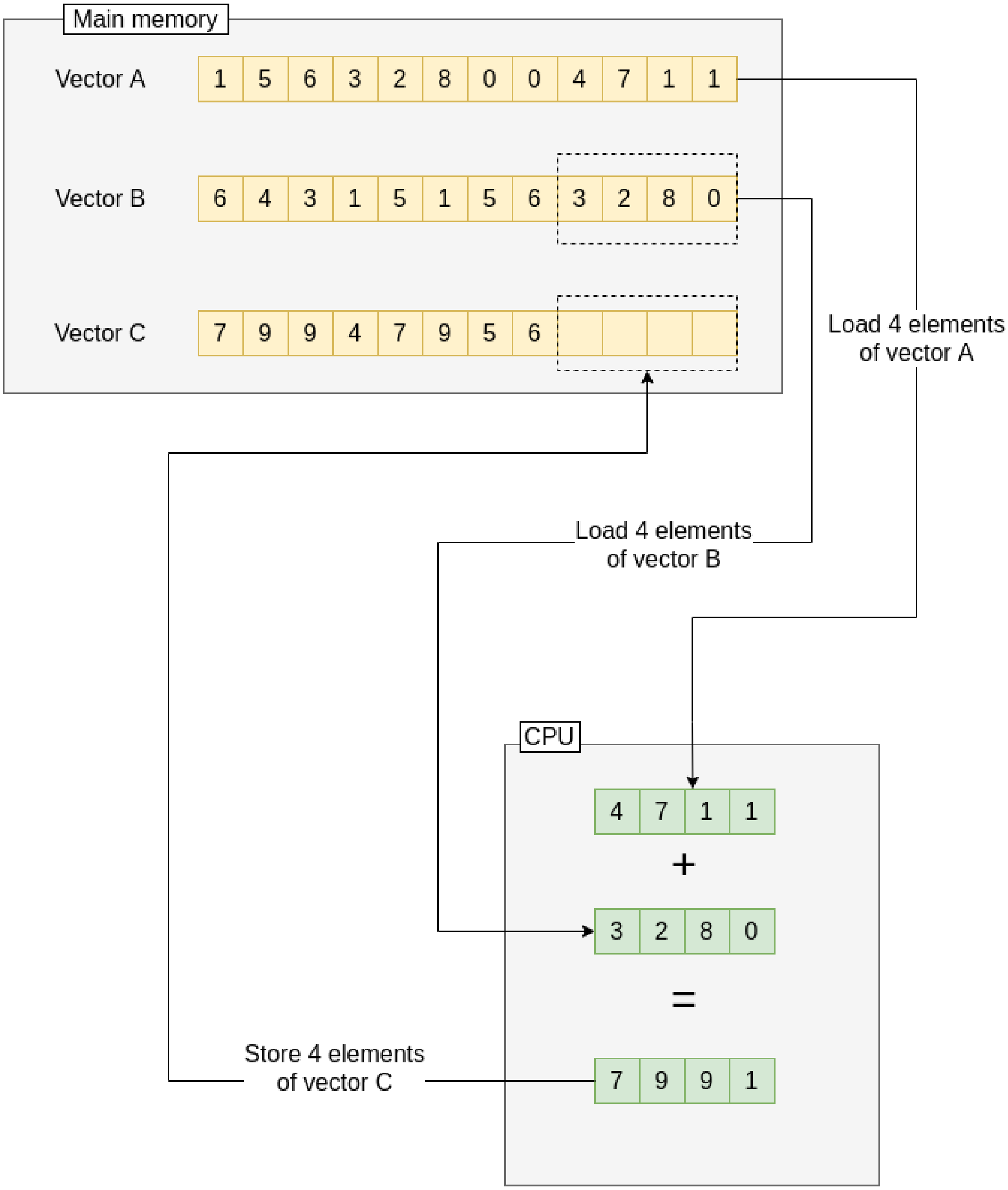
EnlighterJS 3 Syntax Highlighter

```
for (int i = 0; i < n; i+=4) {
double<4> b_val = load<4>(B + i);
double<4> c_val = load<4>(C + i);
double<4> a_val = add<4>(b_val, c_val);
store<4>(a_val, A + i);
}
```

```
for (int i = 0; i < n; i+=4) { double<4> b_val =
load<4>(B + i); double<4> c_val = load<4>(C + i);
double<4> a_val = add<4>(b_val, c_val); store<4>
(a_val, A + i); }
```

```
for (int i = 0; i < n; i+=4) {
    double<4> b_val = load<4>(B + i);
    double<4> c_val = load<4>(C + i);
    double<4> a_val = add<4>(b_val, c_val);
    store<4>(a_val, A + i);
}
```

On line 2 we load four doubles from address `B + i` to register `b_val`. On line 4, we add together two vector registers, `b_val` and `c_val`, each of which contains 4 doubles. On line 5, we store 4 double results to the main memory.



What happens when compiler vectorizes the loop calculating A[i] = B[i] + C[i]

The compilers, when proper switches are applied[3], can automatically vectorize loops which ideally can make them run a few times faster. The compiler analyzes the loop to determine if the vectorization is possible and beneficial. Here are a few prerequisites for successful vectorization:

- **Simple types**: vectorization pays of when the program is running on simple types. Under simple we either mean builtin types like `int`, `double` or `char`; or small classes and structs where each element has a same type like `struct complex { double re; double im;}` or `struct argb {char a; char r; char g; char b;}`

- **The loop is iterating through the array sequentially, i.e. going from 0 to N or from N to 0 with the the increment 1.**[4] If this is not the case, the program is most likely to hit the memory wall and vectorization will not be beneficial there.
- **The loop is countable**: loop is countable if its iteration count can be calculated before the loop begins. An example of a countable loop is a loop going from 0 to N. An example of uncountable loop is a loop calculating the length of the string (by looking for special character `\0`). In the uncountable loop we don’t know how many iterations will the loop have before the loop exits.
- **There are no loop carried dependencies**: the iterations of the loop can be executed independently. A loop calculating `A[i] = B[i] + C[i]` doesn’t have a loop carried dependecies. On the other hand, a loop calculating `A[i] = B[i] + A[i - 1]` has a loop carried dependency: the value of `A[i]` depends on the value of `A[i - 1]`.
- **There is no pointer aliasing in the loop**: if two pointers point to the (partially) overlapping blocks of the memory, we say that the two pointer alias each other. For example, the loop calculating `A[i] = B[i] + C[i]` can be vectorized if there is no pointer aliasing between pointers `A`, `B` and `C`. But, imagine that `double * C = A - 1`. If this is the case, then our code is actually calculating `A[i] = B[i] + A[i -`

1] and this loop has a loop carried dependency. If you are sure there is no pointer aliasing in your code, you can use the __restrict__ keyword to mark the pointers that do not overlap one another in order to promote vectorization.

There are tricks that one can use to vectorize codes that do not completely fulfill the above conditions, but the compilers typically do not employ them. You can use compiler-specific switches to print a vectorization report: the vectorization report should tell you if the loop is vectorized and if not, what prevented the vectorization.

A few tricks related to vectorization

Related to vectorization, there are a few tricks to allow vectorizing some common cases. Here are they:

Reductions

Reductions are very common in source code. They are named reductions because they reduce the values of an array to one value. Calculating a sum of all elements in an array is an example of a reduction. Here is the source code:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

```
double sum = 0.0;
for (int i = 0; i < n; i++) {
sum += a[i];
}
```

```
double sum = 0.0; for (int i = 0; i < n; i++) { sum += a[i]; }
```

```
double sum = 0.0;

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

The problem with vectorizing this loops is that there is a loop carried dependency. In the current iteration we need the value of `sum` calculated in the previous iteration. This is an obstacle to vectorization.

Luckily, there is a simple trick to enable vectorization even then. Instead of having one `sum` variable, we have one `sum` variable for each lane of the vector. We add together lanes independently of one another. And finally, we reduce the vector of sum to one `sum` variable. Written in pseudo-assembler, it looks like this:

Plain text
Copy to clipboard
Open code in new window
EnlighterJS 3 Syntax Highlighter

```
double<4> vec_sum = { 0.0, 0.0, 0.0, 0.0 };
for (int i = 0; i < n; i+=4) {
double<4> a_val = load<4>(a + i);
vec_sum = add<4>(a, vec_sum);
}
sum = 0.0;
for (int i = 0; i < 4; i++) {
sum += vec_sum[i];
}
```

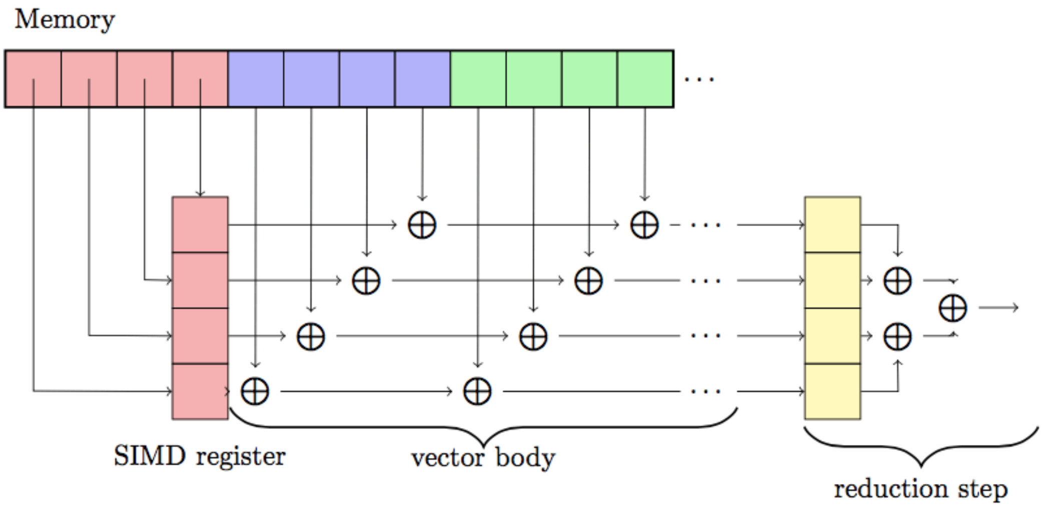
```
double<4> vec_sum = { 0.0, 0.0, 0.0, 0.0 }; for (int i = 0; i < n; i+=4) { double<4> a_val = load<4>(a + i); vec_sum = add<4>(a, vec_sum); } sum = 0.0; for (int i = 0; i < 4; i++) { sum += vec_sum[i]; }
```

```
double<4> vec_sum = { 0.0, 0.0, 0.0, 0.0 };

for (int i = 0; i < n; i+=4) {
    double<4> a_val = load<4>(a + i);
    vec_sum = add<4>(a, vec_sum);
}

sum = 0.0;
for (int i = 0; i < 4; i++) {
    sum += vec_sum[i];
}
```

The variable `vec_sum` is a vector register holding the four values of `sum`, for each lane of the vector (line 1). In the summing loop (lines 3-6) we accumulate the values in the lanes independently. When the calculation is done, we reduce the value of `vec_sum` to a single value `sum` (line 9-11).



Vector reduction. The SIMD register holds the current value of vec_sum. In the reduction step, vec_sum is converted to a single sum value.[5]

This is possible to do because operand `+` is associative, i.e. `(a + b) + c = a + (b + c)`[6].

Vectorization of interleaved data

Vectorization works best when simple data types are stored in the memory consecutively. But often this is not the case. Take for example a complex number, it consists of a real and imaginary part, i.e. `struct complex { double re; double im; };`. In this particular case, the operations we do on the `re` part are not necessarily the same operation we want to do on the `im` part. For example, calculating the square of a complex number is done according to the following formula:

Plain text
Copy to clipboard
Open code in new window
EnlighterJS 3 Syntax Highlighter

```
complex square(complex in) {
complex result;
result.re = in.re * in.re - in.im * in.im;
result.im = 2.0 * in.re * in.im;
return result;
}
```

```
complex square(complex in) { complex result; result.re = in.re * in.re - in.im * in.im; result.im = 2.0 * in.re * in.im; return result; }
```

```
complex square(complex in) {
    complex result;

    result.re = in.re * in.re - in.im * in.im;
    result.im = 2.0 * in.re * in.im;

    return result;
}
```

An array holding complex numbers can be treated as a simple `double` array that stores `re` member on even positions and `im` member on odd positions. When a complex number is loaded into a vector register, it will have `re` member at even lanes and `im` member at odd lanes.

This is the problem for vectorization, as vector instructions can only do the same type of processing on all data in the vector. We want different processing for re and im parts, i.e. we want a different type of processing for odd and even lanes in the vector register.

To vectorize such loops, the program needs to move all re values to one vector register and all im values to another vector registers. Luckily, all vector-enabled processors support interleave and deinterleave instructions (often called permute instructions or shuffle instructions). What this means, after four complex numbers are loaded to two vector registers, the program executes a deinterleave instruction, which moves all the re parts to one vector register and all the im parts to another.

After performing the square on the complex number, but before storing the results back to the main memory, the program executes a interleave instruction which interleaves re and im parts, putting re parts at even lanes and im parts at odd lanes. After this, a simple vector store will store the complex number according to the required memory layout.

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!

Roads rarely taken

There are certain optimizations that the compilers could do, but you will rarely see a compiler doing them. Here we present the three most important according to the author’s opinion, but there are several others as well.

Loop Interchange

Loop interchange is an optimization technique aimed at improving the memory access pattern. A good memory access pattern is crucial for the good performance of your code. Consider the following loop:
Plain text
Copy to clipboard
Open code in new window
EnlighterJS 3 Syntax Highlighter

```
for (int i = 1; i < LEN; i++) {
for (int j = 0; j < LEN; j++) {
b[j][i] = a[j][i] - a[j - 1][i];
}
}
```

for (int i = 1; i < LEN; i++) { for (int j = 0; j < LEN; j++) { b[j][i] = a[j][i] - a[j - 1][i]; } }

```
for (int i = 1; i < LEN; i++) {
    for (int j = 0; j < LEN; j++) {
        b[j][i] = a[j][i] - a[j - 1][i];
    }
}
```

This algorithm goes over an image and calculates the difference between the current pixel and the pixel above it. The algorithm is accessing the image column-wise, i.e. first all the pixels in column 0, then all the pixels in column 1, etc.

This kind of memory access pattern is bad for performance. The image is stored row-wise, and it should be accessed in that manner. The compiler could in principle do the following transformation: exchange the loop over `i` and loop over `j`, this making the loop over `j` the outer loop and loop over `i` the inner loop. Semantically nothing changes, but after the transformation, the image is accessed in a row-wise manner which is much faster and more memory-friendly.

Loop interchange is an optimization technique that can have a dramatic impact on performance. One could expect more than 10x speed improvement with this technique. Generally, it turns out that a good memory access pattern and good memory layout are key to the good performance of your code. If the memory access pattern is good, some compilers will perform better and others will perform worse. But if it is bad, then all compilers will be equally bad.[7]

In LLVM, loop interchange is an experimental compiler step that needs to be enabled explicitly with `-mllvm -enable-loopinterchange` compilation flags.

Loop Distribution

Loop distribution (also called *loop fission*) is an optimization technique that splits a complex non-vectorizable loop into two loops, one which is vectorizable and another one which is not. Consider the following example:

Plain text
Copy to clipboard
Open code in new window
EnlighterJS 3 Syntax Highlighter

```
for (int i = 0; i < n; i++) {
a[i] = a[i - 1] * b[i];
c[i] = a[i] + e[i];
}
```

for (int i = 0; i < n; i++) { a[i] = a[i - 1] * b[i]; c[i] = a[i] + e[i]; }

```
for (int i = 0; i < n; i++) {
    a[i] = a[i - 1] * b[i];
    c[i] = a[i] + e[i];
}
```

The dependency between `a[i]` and `a[i - 1]` on line 2 prevents the vectorization. Splitting the loop into two can help:

Plain text
Copy to clipboard
Open code in new window
EnlighterJS 3 Syntax Highlighter

```
for (int i = 0; i < n; i++) {
a[i] = a[i - 1] * b[i];
}
for (int i = 0; i < n; i++) {
c[i] = a[i] + e[i];
}
```

for (int i = 0; i < n; i++) { a[i] = a[i - 1] * b[i]; } for (int i = 0; i < n; i++) { c[i] = a[i] + e[i]; }

```
for (int i = 0; i < n; i++) {
    a[i] = a[i - 1] * b[i];
}

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

By distributing the loop into two loops, we allow the vectorization of the second loop (since it doesn’t have loop carried dependencies).

There are two reasons why you would want to do loop distribution:

- **Enable vectorization**: if a loop is unvectorizable, splitting the loop into two loops, one of which is vectorizable and the other which is not can help the performance.

- **Avoid register spilling:** in case of large loops with many variables, loop distribution can be used to avoid register spilling.

Loop distribution is a somewhat controversial technique that can result in slowdowns as well. Since we are iterating over the data set two times (in our example we iterate over the array `a` two times), we are putting more stress on the memory subsystem of the computer. The same piece of data has to be brought from the main memory two times, and if there are not enough computations in the fissioned loops, the overall effect can be negative.

In LLVM, loop distribution is an experimental compiler step that has to be enabled with `-mllvm -enable-loop-distribute`. It can be enabled using a compiler pragma as well `pragma clang loop distribute(enable)`.

Loop Fusion

Opposite of loop distribution is loop fusion. With loop fusion we merge together two independent loops. Consider the following example:

Plain text
Copy to clipboard
Open code in new window
EnlighterJS 3 Syntax Highlighter

```
// Find minimum loop
auto min = a[0];
for (int i = 1; i < n; i++) {
if (a[i] < min) min = a[i];
}
// Find maximum loop
auto max = a[0];
for (int i = 1; i < n; i++) {
```

```
if (a[i] > max) max = a[i];
}
```

// Find minimum loop auto min = a[0]; for (int i = 1; i < n; i++) { if (a[i] < min) min = a[i]; } // Find maximum loop auto max = a[0]; for (int i = 1; i < n; i++) { if (a[i] > max) max = a[i]; }

```
// Find minimum loop
auto min = a[0];

for (int i = 1; i < n; i++) {
    if (a[i] < min) min = a[i];
}

// Find maximum loop
auto max = a[0];

for (int i = 1; i < n; i++) {
    if (a[i] > max) max = a[i];
}
```

The first loop finds the minimum of the array, the second loop finds the maximum in the array. The loops are iterating over the same dataset (array `a`) two times.

What this means is that each piece of data has to be brought to memory twice. To relieve the strain on the memory subsystem, a smart compiler can merge these two loops into one loop that looks both for minimum and maximum.

When should I take matters into my own hands?

The answer to this question is not simple, and I plan to cover it in the upcoming post. In the short term, sometimes it is useful to take optimizations into your hands either:

- Giving the compiler enoigh hints so it can perform the optimization job better
- Doing the job of the compiler: many optimizations you can do by hand, at the expense of less readable and less maintainable code.

We talk about this in the next post.

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!

Conclusion

The world of compiler optimizations is a marvelous place and many wonderful things are going on there. As you have seen, the compilers employ some very clever ideas in order to make your code run faster.

But if you have a loop in your code where the speed is critical, it can pay off to understand what are the common optimizations the compiler does, to check the optimization report of the compiler, and to do the necessary tweaks in order to make it run faster.

In the next post, we talk about when you should take matters into your own hands. And in the post after that we talk about `opt-viewer`, a tool that lets you inspect LLVM’s compiler optimizations in a GUI.

1. Scalar value a single value, as opposed to vector value. [

↩️
]

2. Basic block is a segment of instructions with one entry and one exit. If the loop body doesn’t contain any conditional instructions, it will make a basic block [

↩]

3. On GCC and CLANG specifying `-O3` will enable vectorization. Additionally, specifying `-ffast-math` will enable vectorization on some additional loops [

↩
]

4. AVX512 and SVE ISAs support efficient vectorization of loops where the increment is greater than 1 [

↩
]

5. Image source: Demystifying Auto-vectorization in Julia [

↩]

6. For floating-point types, associativity laws do not hold completely because of the limited precision of floating-point types. To enable vectorization in those cases, you would need to provide compiler flags that allow reordering of floating-point operations, such as `-fassociative-math` or `-ffast-math` [

↩
]

7. Reason is simple: the speed of the program doesn’t depend on the quality of emitted instructions, but it depends on the time the program needs to fetch data from the main memory [

]