Lecture 19 — Single Thread Performance, Compiler Optimizations

Patrick Lam
patrick.lam@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 7, 2018

Part I

Single-Thread Performance

"Can you run faster just by trying harder?"

The performance improvements we've seen to date have been leveraging parallelism to improve throughput.

Decreasing latency is trickier—it often requires domain-specific tweaks.

We'll look at one example of decreasing latency today, Stream VByte.

I have a cunning plan...

Even this example leverages parallelism—it uses vector instructions.

But there are some sequential improvements, e.g. Stream VByte takes care to be predictable for the branch predictor.

Problem Context

We can abstract the problem to that of storing a sequence of small integers.

Such sequences are important in the context of inverted indexes, which allow fast lookups by term, and support boolean queries which combine terms.

Here is a list of documents and some terms that they contain:

docid	terms
1	dog, cat, cow
2	cat
3	dog, goat
4	cow, cat, goat

The inverted index looks like this:

term	docs
dog	1, 3
cat	1, 2, 4
cow	1, 4
goat	3, 4

Inverted indexes contain many small integers in their lists.

The deltas are typically small if the list of doc ids is sorted.

VByte

VByte is one of a number of schemes that use a variable number of bytes to store integers.

This makes sense when most integers are small, and especially on today's 64-bit processors.

VByte works like this:

- x between 0 and $2^7 - 1$, e.g. $17 = 0b10001$: $0xxxxxxx$, e.g. 00010001;
- x between 2^7 and $2^{14} - 1$, e.g. $1729 = 0b11011000001$: $1xxxxxxx/0xxxxxxx$, e.g. $11000001/00001101$;
- x between 2^{14} and $2^{21} - 1$: $0xxxxxxx/1xxxxxxx/1xxxxxxx$;
- etc.

That is, the control bit, or high-order bit, is 0 if you have finished representing the integer, and 1 if more bits remain.

It might seem that dealing with variable-byte integers might be harder than dealing fixed-byte integers, and it is.

But there are performance benefits: because we are using fewer bits!

We can fit more information into our limited RAM and cache, and even get higher throughput.

Storing and reading 0s isn't an effective use of resources.

However, a naive algorithm to decode VByte also gives lots of branch mispredictions.

Stream VByte

Stream VByte is a variant of VByte which works using SIMD instructions.

Science is incremental, and Stream VByte builds on earlier work—masked VByte as well as VARINT-GB and VARINT-G8IU.

The innovation in Stream VByte is to store the control and data streams separately.

Stream VByte's control stream uses two bits per integer to represent the size of the integer:

00	1 byte	10	3 bytes
01	2 bytes	11	4 bytes

Each decode iteration reads a byte from the control stream and 16 bytes of data from memory.

It uses a lookup table over the possible values of the control stream to decide how many bytes it needs out of the 16 bytes it has read.

It then uses SIMD instructions to shuffle the bits each into their own integers.

Note that, unlike VByte, Stream VByte uses all 8 bits of each data byte as data.

Stream VByte Example

For instance, if the control stream contains $0b1000\ 1100$, then the data stream contains the following sequence of integer sizes: $3, 1, 4, 1$.

Out of the 16 bytes read, this iteration will use 9 bytes; it advances the data pointer by 9.

It then uses the SIMD “shuffle” instruction to put the decoded integers from the data stream at known positions in the 128-bit SIMD register.

In this case, it pads the first 3-byte integer with 1 byte, then the next 1-byte integer with 3 bytes, etc.

Let's say that the input is `0xf823 e127 2524 9748 1b.. .... .... .....`

The 128-bit output is `0x00f8 23e1/0000 0027/2524 9748/0000/001b`, with the /s denoting separation between outputs.

The shuffle mask is precomputed and, at execution time, read from an array.

The core of the implementation uses three SIMD instructions:

```
uint8_t C = lengthTable[control];
__m128i Data = _mm_loadu_si128 ((__m128i *) databytes);
__m128i Shuf = _mm_loadu_si128(shuffleTable[control]);
Data = _mm_shuffle_epi8(Data, Shuf);
databytes += C; control++;
```

But Does It Work?!

Stream VByte performs better than previous techniques on a realistic input.

Let's discuss how it achieves this performance.

- control bytes are sequential: the processor can always prefetch the next control byte, because its location is predictable;
- data bytes are sequential and loaded at high throughput;
- shuffling exploits the instruction set so that it takes 1 cycle;
- control-flow is regular (executing only the tight loop which retrieves/decodes control and data; there are no conditional jumps).

Part II

Compiler Optimizations

Lunch Time, Already?

"Is there any such thing as a free lunch?"

Compiler optimizations really do feel like a free lunch.

But what does it really mean when you say `-O2`?

We'll see some representative compiler optimizations and discuss how they can improve program performance.

You Do Your Job...

I'll point out cases that stop compilers from being able to optimize your code.

In general, it's better if the compiler automatically does a performance-improving transformation rather than you doing it manually.

It's probably a waste of time for you and it also makes your code less readable.

Many pages on the Internet describe optimizations.

Here's one that contains good examples:

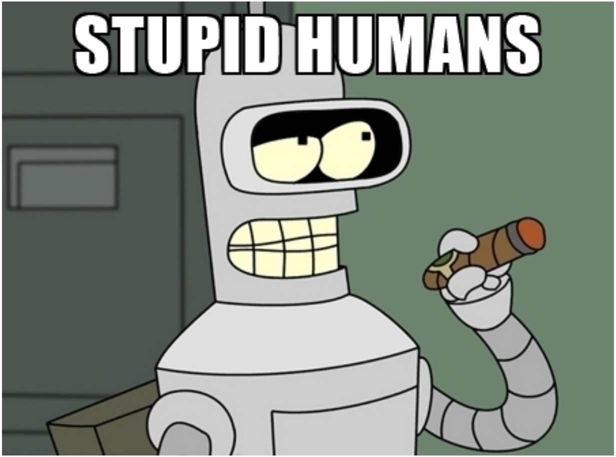
`http://www.digitalmars.com/ctg/ctgOptimizer.html`

You can find a full list of `gcc` options here:

`http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

First of all, “optimization” is a bit of a misnomer.

Compilers generally don’t generate “optimal” code. They generate *better* code.



Optimization Levels

Here's what `-On` means for `gcc`. Other compilers have similar (but not identical) optimization flags.

- `-O0` (default): Fastest compilation time. Debugging works as expected.
- `-O1` (`-O`): Reduce code size and execution time. No optimizations that increase compiliation time.
- `-O2`: All optimizations except space vs. speed tradeoffs.
- `-O3`: All optimizations.
- `-Ofast`: All `-O3` optimizations, plus non-standards compliant optimizations, particularly `-ffast-math`.

Scalar Optimizations: Constant Folding

Tag line: “Why do later something you can do now?”

$$\text{i = 1024 * 1024} \quad \Longrightarrow \quad \text{i = 1048576}$$

Enabled at all optimization levels.

The compiler will not emit code that does the multiplication at runtime.

Common Subexpression Elimination

We can do common subexpression elimination when the same expression `x op y` is computed more than once.

Neither x nor y may change between the two computations.

```
a = (c + d) * y;
b = (c + d) * z;

w = 3;
x = f(); y = x;
z = w + y;
```

Enabled at `-O2`, `-O3` *or with* `-fgcse`

Constant Propagation

Moves constant values from definition to use.

The transformation is valid if there are no redefinitions of the variable between the definition and its use.

In the above example, we can propagate the constant value 3 to its use in `z = w + y`, yielding `z = 3 + y`.

Copy Propagation

A bit more sophisticated than constant propagation—telescopes copies of variables from their definition to their use.

Using it, we can replace the last statement with `z = w + x`.

If we run both constant and copy propagation together, we get `z = 3 + x`.

These scalar optimizations are more complicated in the presence of pointers, e.g. `z = *w + y`.

Dead code elimination: removes code that is guaranteed to not execute.

```
int f(int x) {
  return x * 2;
}
```

```
int g() {
  if (f(5) % 2 == 0)
        {
    // do stuff...
  } else {
    // do other stuff
  }
}
```

The general problem, as with many other compiler problems, is undecidable.

Loop Optimizations

Loop optimizations are particularly profitable when loops execute often.

This is often a win, because programs spend a lot of time looping.

The trick is to find which loops are going to be the important ones.

A loop induction variable is a variable that varies on each iteration of the loop.

The loop variable is definitely a loop induction variable but there may be others.

Induction variable elimination gets rid of extra induction variables.

Scalar Replacement

Scalar replacement replaces an array read `a[i]` occuring multiple times with a single read `temp = a[i]` and references to `temp` otherwise.

It needs to know that `a[i]` won't change between reads.

Sane languages include array bounds checks.

Loop optimizations can eliminate array bounds checks if they can prove that the loop never iterates past the array bounds.

This lets the processor run more code without having to branch as often.

Software pipelining is a synergistic optimization, which allows multiple iterations of a loop to proceed in parallel.

This optimization is also useful for SIMD. Here's an example.

```
for (int i = 0; i < 4;
     ++i)
    f(i)
```

$\Longrightarrow$

```
f(0); f(1); f(2); f(3);
```

Enabled with `-funroll-loops`.

This optimization can give big wins for caches (which are key); it changes the nesting of loops to coincide with the ordering of array elements in memory.

```
for (int i = 0; i < N; ++i)
    for (int j = 0; j < M; ++j)
        a[j][i] = a[j][i] * c
```

$\Longrightarrow$

```
for (int j = 0; j < M; ++j)
    for (int i = 0; i < N; ++i)
        a[j][i] = a[j][i] * c
```

since C is *row-major* (meaning a[1][1] is beside a[1][2]), rather than *column-major*.

Enabled with `-floop-interchange`.

This optimization is like the OpenMP collapse construct; we transform

```
for (int i = 0; i <
    100; ++i)
    a[i] = 4

for (int i = 0; i <
    100; ++i)
    b[i] = 7
```

$\Longrightarrow$

```
for (int i = 0; i < 100; ++i) {
    a[i] = 4
    b[i] = 7
}
```

There's a trade-off between data locality and loop overhead.

Sometimes the inverse transformation, *loop fission*, will improve performance.

Loop-Invariant Code Motion

Also known as *Loop hoisting*, this optimization moves calculations out of a loop.

```
for (int i = 0; i <
     100; ++i) {
    s = x * y;
    a[i] = s * i;
}
```

$\Longrightarrow$

```
s = x * y;
for (int i = 0; i < 100; ++i) {
    a[i] = s * i;
}
```

This reduces the amount of work we have to do for each iteration of the loop.

Miscellaneous Low-Level Optimizations

Some optimizations affect low level code generation; here are two examples.

`gcc` attempts to guess the probability of each branch to best order the code. (For an `if`, fall-through is most efficient. Why?)

This isn't quite an optimization, but you can use `__builtin_expect(expr, value)` to help gcc, if you know the runtime behaviour...

In the Linux Kernel:

```
#define likely(x)       __builtin_expect((x),1)
#define unlikely(x)     __builtin_expect((x),0)
```

Architecture-Specific

`gcc` can also generate code tuned to particular processors.

You can specify this using `-march` and `-mtune`. (`-march` implies `-mtune`).

This will enable specific instructions that not all CPUs support (e.g. SSE4.2). For example, `-march=corei7`.

Good to use on your local machine or your cloud servers, not ideal for code you ship to others.