# Lecture 19 — Single Thread Performance

Patrick Lam
patrick.lam@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 5, 2018

"Can you run faster just by trying harder?"

Performance improvements to date have used parallelism to improve throughput.

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

Today: one example of decreasing latency:
Stream VByte.

# I have a cunning plan...

Even Stream VByte uses parallelism:
vector instructions.

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

# Inverted Indexes (like it's CS 137 again!)

Abstractly: store a sequence of small integers.

Why Inverted indexes?

allow fast lookups by term;
support boolean queries combining terms.

# Dogs, cats, cows, goats. In ur documents.

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

# Inverting the Index

Here's the index and the inverted index:

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

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

Inverted indexes contain many small integers.

Deltas typically small if doc ids are sorted.

# Storing inverted index lists: VByte

VByte uses a variable number of bytes to store integers.

Why? Most integers are small, 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.

Control bit, or high-order bit, is:
0 once done representing the int,
1 if more bits remain.

# Why VByte Helps

Isn't dealing with variable-byte integers harder?

- Yup!

But perf improves:

- We are using fewer bits!

We fit more information into RAM and cache, and can get higher throughput. (think inlining)

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

However, a naive algorithm to decode VByte gives branch mispredicts.

Stream VByte: a variant of VByte using SIMD.

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

Innovation in Stream VByte:
*store the control and data streams separately*.

# Control Stream

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 |

# Decoding Stream VByte

Per decode iteration:

reads 1 byte from the control stream,
and 16 bytes of data.

Lookup table on control stream byte: decide how many bytes it needs out of the 16 bytes it has read.

SIMD instructions:

shuffle the bits each into their own integers.

Unlike VByte,
Stream VByte uses all 8 bits of data bytes as data.

Say 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 uses 9 bytes;
$\Rightarrow$ it advances the data pointer by 9.

The SIMD "shuffle" instruction puts decoded integers from data stream at known positions in the 128-bit SIMD register.

Pad the first 3-byte integer with 1 byte, then the next 1-byte integer with 3 bytes, etc.

Say the data input is:
`0xf823 e127 2524 9748 1b.. .... .... .....`

The 128-bit output is:
`0x00f8 23e1/0000 0027/2524 9748/0000/001b`
/s denote separation between outputs.

Shuffle mask is precomputed and 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.

Why?

- control bytes are sequential:
  CPU 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:
  takes 1 cycle;
- control-flow is regular
  (tight loop which retrieves/decodes control & data;
  no conditional jumps).