Vectorized VByte Decoding

Jeff Plaisance, Nathan Kurz, Daniel Lemire

Inverted Indexes

Inverted Index

- Like index in the back of a book
- words = terms, page numbers = doc ids
- Term list is sorted
- Doc list for each term is sorted

Standard Index

doc id	query	country	impressions	clicks
0	software	Canada	10	1
1	blank	Canada	10	0
2	sales	US	5	0
3	software	US	8	1
4	blank	US	10	1

Constructing an Inverted Index

	query			country		impression			clicks	
doc id	blank	sales	software	Canada	US	5	8	10	0	1
0			✔	✔				✔		✔
1	✔			✔				✔	✔	
2		✔			✔	✔			✔	
3			✔		✔		✔			✔
4	✔				✔			✔		✔

Constructing an Inverted Index

field	term	0	1	2	3	4
query	blank		✔			✔
	sales			✔		
	software	✔			✔	
country	Canada	✔	✔			
	US			✔	✔	✔
impressions	5			✔		
	8				✔	
	10	✔	✔			✔
clicks	0		✔	✔		
	1	✔			✔	✔

Inverted Index

field	term	doc list
query	blank	1, 4
	sales	2
	software	0, 3
country	Canada	0, 1
	US	2, 3, 4
impressions	5	2
	8	3
	10	0, 1, 4
clicks	0	1, 2
	1	0, 3, 4

Inverted Indexes

Allow you to:

- Quickly find all documents containing a term
- Intersect several terms to perform boolean queries

Inverted Index Optimizations

- Compressed data structures
  - Better use of RAM and processor cache
  - Better use of memory bandwidth
  - Increased CPU usage and time
- Optimizations matter!

Delta / VByte Encoding

- Doc id lists are sorted
- Delta between a doc id and the previous doc id is sufficient
- Deltas are usually small integers

Delta Encoding

field	term	doc list
query	nursing	34, 86, 247, 301, 674, 714

Delta Encoding

field	term	doc list
query	nursing	34, 86, 247, 301, 674, 714
		34, 52, 161, 54, 373, 40

Small Integer Compression

- Golomb/Rice
- VByte (or Varint)
- Binary Packing
- PForDelta

Small Integer Compression

- Golomb/Rice
- **VByte**
- Bit Packing
- PForDelta

VByte Encoding

9838

VByte Encoding

9838

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	1	1	0	0	1	1	0	1	1	1	0

VByte Encoding

9838

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**		**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	0	1	0	0	1	1	0		0	1	1	0	1	1	1	0

VByte Encoding

9838

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**0**

VByte Encoding

9838

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	1	1	0	0	**1**	**1**	**0**	**1**	**1**	**1**	**0**

VByte Encoding

9838

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	1	1	0	0	**1**	**1**	**0**	**1**	**1**	**1**	**0**

?	**1**	**1**	**0**	**1**	**1**	**1**	**0**

VByte Encoding

9838

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	1	1	0	0	**1**	**1**	**0**	**1**	**1**	**1**	**0**

?	1	1	0	1	1	1	0

VByte Encoding

9838

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	1	1	0	0	1	1	0	1	1	1	0

?	1	1	0	1	1	1	0

VByte Encoding

9838

0	0	0	0	0	0	0	0		0	0	0	0	0	0	0	0
0	0	1	0	0	1	1	0		0	1	1	0	1	1	1	0

1	1	1	0	1	1	1	0

VByte Encoding

9838

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	**1**	**0**	**0**	**1**	**1**	**0**	**0**	1	1	0	1	1	1	0

1	1	1	0	1	1	1	0

VByte Encoding

9838

0	0	0	0	0	0	0	0
0	0	**1**	**0**	**0**	**1**	**1**	**0**

0	0	0	0	0	0	0	0
0	1	1	0	1	1	1	0

1	1	1	0	1	1	1	0
?	**1**	**0**	**0**	**1**	**1**	**0**	**0**

VByte Encoding

9838

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	**1**	**0**	**0**	**1**	**1**	**0**	**0**	1	1	0	1	1	1	0

1	1	1	0	1	1	1	0
?	1	0	0	1	1	0	0

VBytes Encoding

9838

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	1	0	0	1	1	0	0	1	1	0	1	1	1	0

1	1	1	0	1	1	1	0
?	1	0	0	1	1	0	0

VByte Encoding

9838

0	0	0	0	0	0	0	0		0	0	0	0	0	0	0	0
0	0	1	0	0	1	1	0		0	1	1	0	1	1	1	0

1	1	1	0	1	1	1	0
0	1	0	0	1	1	0	0

VByte Encoding

9838

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	1	1	0	0	1	1	0	1	1	1	0

1	**1**	**1**	**0**	**1**	**1**	**1**	**0**
0	**1**	**0**	**0**	**1**	**1**	**0**	**0**

VByte

Pros:

- Compression
- Can fit more of index in RAM
- Higher information throughput per byte read from disk

VByte

Cons:

- Decodes one byte at a time
- Lots of branch mispredictions
- Not fast to decode
- Largest ints expand to 5 bytes

Vectorized VByte Decoding

Optimized decoder implemented using x86_64 intrinsics

Vectorized VByte Decoding

```
01001010 11001000 01110001 01001110
10011011 01101010 10110101 00010111
01110110 10001101 10110011 11000001
```

Vectorized VByte Decoding

```
01001010 11001000 01110001 01001110
10011011 01101010 10110101 00010111
01110110 10001101 10110011 11000001
```

`pmovmskb:` Extract top bit of each byte

Vectorized VByte Decoding

```
01001010 11001000 01110001 01001110
10011011 01101010 10110101 00010111
01110110 10001101 10110011 11000001
```

`pmovmskb:` Extract top bit of each byte

0100101001001111

`0100101001111`

Pattern of leading bits determines:

- how many varints to decode
- sizes and offsets of varints
- length of longest varint in bytes
- number of bytes to consume

`010010100111`

Pattern of leading bits determines:

- how many varints to decode
- sizes and offsets of varints
- length of longest varint in bytes
- number of bytes to consume

`010010100111`

Pattern of leading bits determines:

- how many varints to decode
- sizes and offsets of varints
- length of longest varint in bytes
- number of bytes to consume

`010010100111`

Pattern of leading bits determines:

- how many varints to decode
- sizes and offsets of varints
- length of longest varint in bytes
- number of bytes to consume

`010010100111`

Pattern of leading bits determines:

- how many varints to decode
- sizes and offsets of varints
- length of longest varint in bytes
- number of bytes to consume

010010100111

Pattern of leading bits determines:

- how many varints to decode
- sizes and offsets of varints
- length of longest varint in bytes
- number of bytes to consume

010010100111

Pattern of leading bits determines:

- how many varints to decode
- sizes and offsets of varints
- length of longest varint in bytes
- number of bytes to consume

`010010100111`

Pattern of leading bits determines:

- how many varints to decode
- sizes and offsets of varints
- length of longest varint in bytes
- number of bytes to consume

010010100111

Pattern of leading bits determines:

- how many varints to decode
- sizes and offsets of varints
- length of longest varint in bytes
- number of bytes to consume

`0100101001`**`00`**~~`111`~~

Decoding options for:

- sixteen 1 byte varints
- six 1-2 byte varints
- four 1-3 byte varints
- two 1-5 byte varints

0100101000111

Decoding options for:

- sixteen 1 byte varints - special case
- six 1-2 byte varints - 2^6, 64 possibilities
- four 1-3 byte varints - 3^4, 81 possibilities
- two 1-5 byte varints - 5^2, 25 possibilities

170 total possibilities

`01001010``111`

Data Distribution:

- Longer doc id lists are necessarily composed of smaller deltas
- Most deltas in real datasets (ClueWeb09, Indeed’s internal datasets) fall into 1 byte case or 1-2 byte case

Most Significant Bit Decoding

- We separate most significant bit decoding from integer decoding
- Reduces duplicate most significant bit decoding work if we don't consume all 12 bytes
- Better instruction level parallelism

010010100111

- If most significant bits of next 16 bytes are all 0, handle sixteen 1 byte ints case
- Otherwise lookup most significant bits of next 12 bytes in 4096 entry lookup table

010010100111

Lookup table contains:

- Shuffle vector index from 0-169 representing which possibility we are decoding
- Number of bytes of input that will be consumed

`01001010` `0111`

Branch on shuffle vector index to determine which case we are decoding

- 0-63 - six 1-2 byte ints
- 64-144 - four 1-3 byte ints
- 145-169 - two 1-5 byte ints

Six 1-2 Byte Ints

```
01001010 11001000 01110001 01001110
10011011 01101010 10110101 00010111
01110110 10001101 10110011 11000001
```

Decode 6 varints from 9 bytes

Expected Positions

Six 1-2 byte ints

2	1	2	1	2	1	2	1	2	1	2	1	0	0	0	0

Four 1-3 byte ints

0	3	2	1	0	3	2	1	0	3	2	1	0	3	2	1

Two 1-5 byte ints

1	5	0	4	0	3	0	2	1	5	0	4	0	3	0	2

Six 1-2 Byte Ints

```
01001010 11001000 01110001 01001110
10011011 01101010 10110101 00010111
01110110 10001101 10110011 11000001
```

Pad out 1 byte ints to 2 bytes

Six 1-2 Byte Ints

```
01001010 00000000    11001000 01110001
01001110 00000000    10011011 01101010
10110101 00010111    01110110 00000000
```

Pad out 1 byte ints to 2 bytes

Shuffle input

- Use index to lookup appropriate shuffle vector
- Shuffle input bytes to get them in the expected positions

pshufb

```
for (i = 0; i < 16; i++) {
    if (mask[i] & 0x80) {
        dest[i] = 0;
    } else {
        dest[i] = src[mask[i] & 0xF];
    }
}
```

pshufb

src

DE	DF	27	E3	7C	A9	60	55	1C	EA	45	56	A6	43	C9	48

mask

0	11	2	-1	12	-1	13	4	0	10	2	6	4	3	13	5

dest

pshufb

src

DE	DF	27	E3	7C	A9	60	55	1C	EA	45	56	A6	43	C9	48

mask

0	11	2	-1	12	-1	13	4	0	10	2	6	4	3	13	5

dest

pshufb

src

DE	DF	27	E3	7C	A9	60	55	1C	EA	45	56	A6	43	C9	48

mask

0	11	2	-1	12	-1	13	4	0	10	2	6	4	3	13	5

dest

pshufb

src

DE	DF	27	E3	7C	A9	60	55	1C	EA	45	56	A6	43	C9	48

mask

0	11	2	-1	12	-1	13	4	0	10	2	6	4	3	13	5

dest

DE															

pshufb

src

DE	DF	27	E3	7C	A9	60	55	1C	EA	45	56	A6	43	C9	48

mask

0	11	2	-1	12	-1	13	4	0	10	2	6	4	3	13	5

dest

DE															

pshufb

src

DE	DF	27	E3	7C	A9	60	55	1C	EA	45	56	A6	43	C9	48

mask

0	11	2	-1	12	-1	13	4	0	10	2	6	4	3	13	5

dest

DE															

pshufb

src

DE	DF	27	E3	7C	A9	60	55	1C	EA	45	56	A6	43	C9	48

mask

0	11	2	-1	12	-1	13	4	0	10	2	6	4	3	13	5

dest

DE	56														

pshufb

src

DE	DF	27	E3	7C	A9	60	55	1C	EA	45	56	A6	43	C9	48

mask

0	11	2	-1	12	-1	13	4	0	10	2	6	4	3	13	5

dest

DE	56														

pshufb

src

DE	DF	27	E3	7C	A9	60	55	1C	EA	45	56	A6	43	C9	48

mask

0	11	2	-1	12	-1	13	4	0	10	2	6	4	3	13	5

dest

DE	56														

pshufb

src

DE	DF	27	E3	7C	A9	60	55	1C	EA	45	56	A6	43	C9	48

mask

0	11	2	-1	12	-1	13	4	0	10	2	6	4	3	13	5

dest

DE	56	27													

pshufb

src

DE	DF	27	E3	7C	A9	60	55	1C	EA	45	56	A6	43	C9	48

mask

0	11	2	-1	12	-1	13	4	0	10	2	6	4	3	13	5

dest

DE	56	27													

pshufb

src

DE	DF	27	E3	7C	A9	60	55	1C	EA	45	56	A6	43	C9	48

mask

0	11	2	-1	12	-1	13	4	0	10	2	6	4	3	13	5

dest

DE	56	27	0												

pshufb

src

DE	DF	27	E3	7C	A9	60	55	1C	EA	45	56	A6	43	C9	48

mask

0	11	2	-1	12	-1	13	4	0	10	2	6	4	3	13	5

dest

DE	56	27	0												

pshufb

src

DE	DF	27	E3	7C	A9	60	55	1C	EA	45	56	A6	43	C9	48

mask

0	11	2	-1	12	-1	13	4	0	10	2	6	4	3	13	5

dest

DE	56	27	0												

pshufb

src

DE	DF	27	E3	7C	A9	60	55	1C	EA	45	56	A6	43	C9	48

mask

0	11	2	-1	12	-1	13	4	0	10	2	6	4	3	13	5

dest

DE	56	27	0	A6											

pshufb

src

DE	DF	27	E3	7C	A9	60	55	1C	EA	45	56	A6	43	C9	48

mask

0	11	2	-1	12	-1	13	4	0	10	2	6	4	3	13	5

dest

DE	56	27	0	A6											

pshufb

src

DE	DF	27	E3	7C	A9	60	55	1C	EA	45	56	A6	43	C9	48

mask

0	11	2	-1	12	-1	13	4	0	10	2	6	4	3	13	5

dest

DE	56	27	0	A6	0										

pshufb

src

DE	DF	27	E3	7C	A9	60	55	1C	EA	45	56	A6	43	C9	48

mask

0	11	2	-1	12	-1	13	4	0	10	2	6	4	3	13	5

dest

DE	56	27	0	A6	0										

pshufb

src

DE	DF	27	E3	7C	A9	60	55	1C	EA	45	56	A6	43	C9	48

mask

0	11	2	-1	12	-1	13	4	0	10	2	6	4	3	13	5

dest

DE	56	27	0	A6	0										

pshufb

src

DE	DF	27	E3	7C	A9	60	55	1C	EA	45	56	A6	43	C9	48

mask

0	11	2	-1	12	-1	13	4	0	10	2	6	4	3	13	5

dest

DE	56	27	0	A6	0	43									

pshufb

src

DE	DF	27	E3	7C	A9	60	55	1C	EA	45	56	A6	43	C9	48

mask

0	11	2	-1	12	-1	13	4	0	10	2	6	4	3	13	5

dest

DE	56	27	0	A6	0	43	7C	DE	45	27	60	7C	E3	43	A9

Shuffle input

- Use index to lookup appropriate shuffle vector
- Shuffle input bytes to get them in the expected positions

Six 1-2 Byte Ints

```
01001010 00000000   11001000 01110001
01001110 00000000   10011011 01101010
10110101 00010111   01110110 00000000
```

Reverse bytes in 2 byte varints

*not actually necessary since x86 is little endian

Six 1-2 Byte Ints

```
00000000 01001010  01110001 11001000
00000000 01001110  01101010 10011011
00010111 10110101  00000000 01110110
```

Reverse bytes in 2 byte varints

*not actually necessary since x86 is little endian

Six 1-2 Byte Ints

```
00000000 01001010  01110001 11001000
00000000 01001110  01101010 10011011
00010111 10110101  00000000 01110110
```

Mask out leading purple 1's

Six 1-2 Byte Ints

```
00000000 01001010  01110001 01001000
00000000 01001110  01101010 00011011
00010111 00110101  00000000 01110110
```

Mask out leading purple 1's

Six 1-2 Byte Ints

00000000	01001010	01110001	01001000
00000000	01001110	01101010	00011011
00010111	00110101	00000000	01110110

Shift top bytes of each varint 1 bit right (mask/shift/or)

Six 1-2 Byte Ints

```
000000000 01001010   000111000 11001000
000000000 01001110   000110101 00011011
000001011 10110101   000000000 01110110
```

Shift top bytes of each varint 1 bit right (mask/shift/or)

Six 1-2 Byte Ints

```
00000000 01001010  00111000 11001000
00000000 01001110  00110101 00011011
00001011 10110101  00000000 01110110
```

Done!

Four 1-3 Byte Ints

```
11101110 00011101  11110101 11101101
01111001 11111000  01101001 00100001
00001011 10110101  10111001 01110110
```

Four 1-3 Byte Ints

```
11101110 00011101   11110101 11101101
01111001 11111000   01101001 00100001
00001011 10110101   10111001 01110110
```

101101000110

Four 1-3 Byte Ints

```
11101110 00011101   11110101 11101101
01111001 11111000   01101001 00100001
00001011 10110101   10111001 01110110
```

1011101000110

Four 1-3 Byte Ints

```
11101110 00011101   11110101 11101101
01111001 11111000   01101001 00100001
00001011 10110101   10111001 01110110
```

1011010000110

Four 1-3 Byte Ints

```
11101110 00011101   11110101 11101101
01111001 11111000   01101001 00100001
00001011 10110101   10111001 01110110
```

1011101000110 0

Four 1-3 Byte Ints

```
11101110 00011101  11110101 11101101
01111001 11111000  01101001 00100001
00001011 10110101  10111001 01110110
```

101101000110

Four 1-3 Byte Ints

```
11101110 00011101   11110101 11101101
01111001 11111000   01101001 00100001
00001011 10110101   10111001 01110110
```

101101000110

Four 1-3 Byte Ints

```
11101110 00011101   11110101 11101101
01111001 11111000   01101001 00100001
00001011 10110101   10111001 01110110
```

Decode 4 varints from 8 bytes

Four 1-3 Byte Ints

```
11101110 00011101   11110101 11101101
01111001 11111000   01101001 00100001
00001011 10110101   10111001 01110110
```

Pad ints to 4 bytes

Four 1-3 Byte Ints

```
11101110 00011101  00000000 00000000
11110101 11101101  01111001 00000000
11111000 01101001  00000000 00000000
00100001 00000000  00000000 00000000
```

Pad ints to 4 bytes

Four 1-3 Byte Ints

```
00000000 00000000 00011101 11101110
00000000 01111001 11101101 11110101
00000000 00000000 01101001 11111000
00000000 00000000 00000000 00100001
```

Reverse bytes

*not actually necessary since x86 is little endian

Four 1-3 Byte Ints

```
00000000 00000000  00011101 11101110
00000000 01111001  11101101 11110101
00000000 00000000  01101001 11111000
00000000 00000000  00000000 00100001
```

Clear top bit of each byte

Four 1-3 Byte Ints

```
00000000 00000000  00011101 01101110
00000000 01111001  01101101 01110101
00000000 00000000  01101001 01111000
00000000 00000000  00000000 00100001
```

Clear top bit of each byte

Four 1-3 Byte Ints

00000000	00000000	00011101	01101110
00000000	01111001	01101101	01110101
00000000	00000000	01101001	01111000
00000000	00000000	00000000	00100001

Shift 2nd least significant bytes over by 1 bit

(mask/shift/or)

Four 1-3 Byte Ints

```
00000000 00000000  00001110 11101110
00000000 01111001  00110110 11110101
00000000 00000000  00110100 11111000
00000000 00000000  00000000 00100001
```

Shift 2nd least significant bytes over by 1 bit

(mask/shift/or)

Four 1-3 Byte Ints

```
00000000 00000000 00001110 11101110
00000000 01111001 00110110 11110101
00000000 00000000 00110100 11111000
00000000 00000000 00000000 00100001
```

Shift 3rd least significant bytes over by 2 bits

(mask/shift/or)

Four 1-3 Byte Ints

```
00000000 00000000  00001110 11101110
00000000 00011110  01110110 11110101
00000000 00000000  00110100 11111000
00000000 00000000  00000000 00100001
```

Shift 3rd least significant bytes over by 2 bits

(mask/shift/or)

Four 1-3 Byte Ints

```
00000000 00000000  00001110 11101110
00000000 00011110  01110110 11110101
00000000 00000000  00110100 11111000
00000000 00000000  00000000 00100001
```

Done!

Two 1-5 Byte Ints

```
11101110 10011101   11110101 11101101
00000011 11111000   11101001 10100001
00001011 10110101   10111001 01110110
```

111101110110

Two 1-5 Byte Ints

```
11101110 10011101   11110101 11101101
00000011 11111000   11101001 10100001
00001011 10110101   10111001 01110110
```

11110111101100

Two 1-5 Byte Ints

```
11101110 10011101   11110101 11101101
00000011 11111000   11101001 10100001
00001011 10110101   10111001 01110110
```

1111011110110

Two 1-5 Byte Ints

```
11101110 10011101   11110101 11101101
00000011 11111000   11101001 10100001
00001011 10110101   10111001 01110110
```

```
1111011100110
```

Two 1-5 Byte Ints

```
11101110 10011101  11110101 11101101
00000011 11111000  11101001 10100001
00001011 10110101  10111001 01110110
```

Decode 2 varints from 9 bytes

Two 1-5 Byte Ints

```
11101110 10011101   11110101 11101101
00000011 11111000   11101001 10100001
00001011 10110101   10111001 01110110
```

- Could handle the same way as other cases
- Would require 5 AND operations, 4 shift operations, and 4 OR operations

Two 1-5 Byte Ints

```
11101110 10011101   11110101 11101101
00000011 11111000   11101001 10100001
00001011 10110101   10111001 01110110
```

- We can simulate shifting by different amounts with multiplication
- Only needs 1 multiplication, 1 shift, 1 OR, 1 shuffle

Two 1-5 Byte Ints

```
11101110 10011101   11110101 11101101
00000011 11111000   11101001 10100001
00001011 10110101   10111001 01110110
```

Two 1-5 byte ints

1	**5**	**0**	**4**	**0**	**3**	**0**	**2**	**1**	**5**	**0**	**4**	**0**	**3**	**0**	**2**

Treat SIMD register as eight 16 bit registers, loading 1 byte into each. First byte doesn't need to be shifted.

Two 1-5 Byte Ints

```
11101110 10011101  11110101 11101101
00000011 11111000  11101001 10100001
00001011 10110101  10111001 01110110

00000000 00000000  00000000 00000000
00000000 00000000  00000000 00000000
```

Two 1-5 Byte Ints

```
11101110 10011101  11110101 11101101
00000011 11111000  11101001 10100001
00001011 10110101  10111001 01110110

11101110 00000000  00000000 00000000
00000000 00000000  00000000 00000000
```

Two 1-5 Byte Ints

```
11101110 10011101  11110101 11101101
00000011 11111000  11101001 10100001
00001011 10110101  10111001 01110110

11101110 00000000  00000000 00000000
00000000 00000000  00000000 10011101
```

Two 1-5 Byte Ints

```
11101110 10011101  11110101 11101101
00000011 11111000  11101001 10100001
00001011 10110101  10111001 01110110

11101110 00000000  00000000 00000000
00000000 11110101  00000000 10011101
```

Two 1-5 Byte Ints

```
11101110 10011101  11110101 11101101
00000011 11111000  11101001 10100001
00001011 10110101  10111001 01110110

11101110 00000000  00000000 11101101
00000000 11110101  00000000 10011101
```

Two 1-5 Byte Ints

```
11101110 10011101  11110101 11101101
00000011 11111000  11101001 10100001
00001011 10110101  10111001 01110110

11101110 00000011  00000000 11101101
00000000 11110101  00000000 10011101
```

Two 1-5 Byte Ints

```
11101110 00000011
00000000 11101101
00000000 11110101
00000000 10011101
```

Clear top bit of each byte

Two 1-5 Byte Ints

```
01101110 00000011
00000000 01101101
00000000 01110101
00000000 00011101
```

Clear top bit of each byte

Two 1-5 Byte Ints

```
01101110 00000011 * 16  (<< 4)
00000000 01101101 * 32  (<< 5)
00000000 01110101 * 64  (<< 6)
00000000 00011101 * 128 (<< 7)
```

Multiply to shift bits into place

Two 1-5 Byte Ints

```
01101110 00000011 * 16  (<< 4)
00000000 01101101 * 32  (<< 5)
00000000 01110101 * 64  (<< 6)
00000000 00011101 * 128 (<< 7)
```

Multiply to shift bits into place

Two 1-5 Byte Ints

```
11100000 00110000 * 16  (<< 4)
00001101 10100000 * 32  (<< 5)
00011101 01000000 * 64  (<< 6)
00001110 10000000 * 128 (<< 7)
```

Multiply to shift bits into place

Two 1-5 Byte Ints

```
11100000 00110000   00001101 10100000
00011101 01000000   00001110 10000000
```

Two 1-5 Byte Ints

`11100000 00110000   00001101 10100000`
`00011101 01000000   00001110 10000000`

Left shift everything by 8 bits

Two 1-5 Byte Ints

```
11100000 00110000  00001101 10100000
00011101 01000000  00001110 10000000
```

Left shift everything by 8 bits

```
00110000 00001101  10100000 00011101
01000000 00001110  10000000 00000000
```

Two 1-5 Byte Ints

```
11100000 00110000   00001101 10100000
00011101 01000000   00001110 10000000
```

Bitwise OR pre-shifted and shifted registers

```
00110000 00001101  10100000 00011101
01000000 00001110  10000000 00000000
```

Two 1-5 Byte Ints

```
11100000 00110000  00001101 10100000
00011101 01000000  00001110 10000000
```

Bitwise OR pre-shifted and shifted registers

```
00110000 00001101  10100000 00011101
01000000 00001110  10000000 00000000
```

Two 1-5 Byte Ints

```
11110000 00110000  00001101 10100000
00011101 01000000  00001110 10000000
```

Bitwise OR pre-shifted and shifted registers

```
00110000 00001101  10100000 00011101
01000000 00001110  10000000 00000000
```

Two 1-5 Byte Ints

```
11110000 00110000   00001101 10100000
00011101 01000000   00001110 10000000
```

Bitwise OR pre-shifted and shifted registers

```
00110000 00001101  10100000 00011101
01000000 00001110  10000000 00000000
```

Two 1-5 Byte Ints

```
11110000 00111101  10101101 10100000
00011101 01000000  00001110 10000000
```

Bitwise OR pre-shifted and shifted registers

```
00110000 00001101  10100000 00011101
01000000 00001110  10000000 00000000
```

Two 1-5 Byte Ints

```
11110000 00111101  10101101 10100000
00011101 01000000  00001110 10000000
```

Bitwise OR pre-shifted and shifted registers

```
00110000 00001101  10100000 00011101
01000000 00001110  10000000 00000000
```

Two 1-5 Byte Ints

```
11110000 00111101  10101101 10111101
01011101 01000000  00001110 10000000
```

Bitwise OR pre-shifted and shifted registers

```
00110000 00001101  10100000 00011101
01000000 00001110  10000000 00000000
```

Two 1-5 Byte Ints

```
11110000 00111101  10101101 10111101
01011101 01000000  00001110 10000000
```

Bitwise OR pre-shifted and shifted registers

```
00110000 00001101  10100000 00011101
01000000 00001110  10000000 00000000
```

Two 1-5 Byte Ints

```
11110000 00111101  10101101 10111101
01011101 01001110  10001110 10000000
```

Bitwise OR pre-shifted and shifted registers

```
00110000 00001101  10100000 00011101
01000000 00001110  10000000 00000000
```

Two 1-5 Byte Ints

```
11110000 00111101  10101101 10111101
01011101 01001110  10001110 10000000
```

Extract result from every other byte

Two 1-5 Byte Ints

```
11110000 00111101   10101101 10111101
01011101 01001110   10001110 10000000
```

Extract result from every other byte

Two 1-5 Byte Ints

```
11110000 00111101  10101101 10111101
01011101 01001110  10001110 10000000
```

Extract result from every other byte

Two 1-5 Byte Ints

```
00110000 00111101  10101101 10111101
01011101 01001110  10001110 10000000
```

Extract result from every other byte

Two 1-5 Byte Ints

```
00110000 00111101  10101101 10111101
01011101 01001110  10001110 10000000
```

Extract result from every other byte

Two 1-5 Byte Ints

```
00110000 00111101  10101101 10111101
01011101 01001110  10001110 10000000
```

Extract result from every other byte

Two 1-5 Byte Ints

```
00110000 00111101  10101101 10111101
01011101 01001110  10001110 10000000
```

Extract result from every other byte

Two 1-5 Byte Ints

```
00110000 00111101  10101101 10111101
01011101 01001110  10001110 10000000
```

Extract result from every other byte

Two 1-5 Byte Ints

```
00110000 00111101  10101101 10111101
01011101 01001110  10001110 10000000
```

Extract result from every other byte

Two 1-5 Byte Ints

```
00110000 00111101  10101101 10111101
01011101 01001110  10001110 10000000
```

Extract result from every other byte

Two 1-5 Byte Ints

```
00110000 00111101  10101101 10111101
01011101 01001110  10001110 10000000
```

OR in low 7 bits of least significant byte

(remember that we stored it in most significant byte position originally)

Two 1-5 Byte Ints

```
00110000 00111101  10101101 10111101
01011101 01001110  10001110 11101110
```

OR in low 7 bits of least significant byte

(remember that we stored it in most significant byte position originally)

Two 1-5 Byte Ints

```
00111101 10111101  01001110 11101110
```

Final result!

Two 1-5 Byte Ints

```
00111101 10111101  01001110 11101110
```

Final result!

Checking my work against initial varint:

```
11101110 10011101  11110101 11101101
00000011
```

Two 1-5 Byte Ints

`00111101 10111101  01001110 11101110`

Final result!

Checking my work against initial varint:

`11101110 10011101  11110101 11101101`
`00000011`

Two 1-5 Byte Ints

`00111101 10111101  01001110 11101110`

Final result!

Checking my work against initial varint:

`11101110 10011101  11110101 11101101`
`00000011`

Two 1-5 Byte Ints

`00111101 10111101  01001110 11101110`

Final result!

Checking my work against initial varint:

`11101110 10011101  11110101 11101101`
`00000011`

Two 1-5 Byte Ints

```
00111101 10111101  01001110 11101110
```

Final result!

Checking my work against initial varint:

```
11101110 10011101  11110101 11101101
00000011
```

Two 1-5 Byte Ints

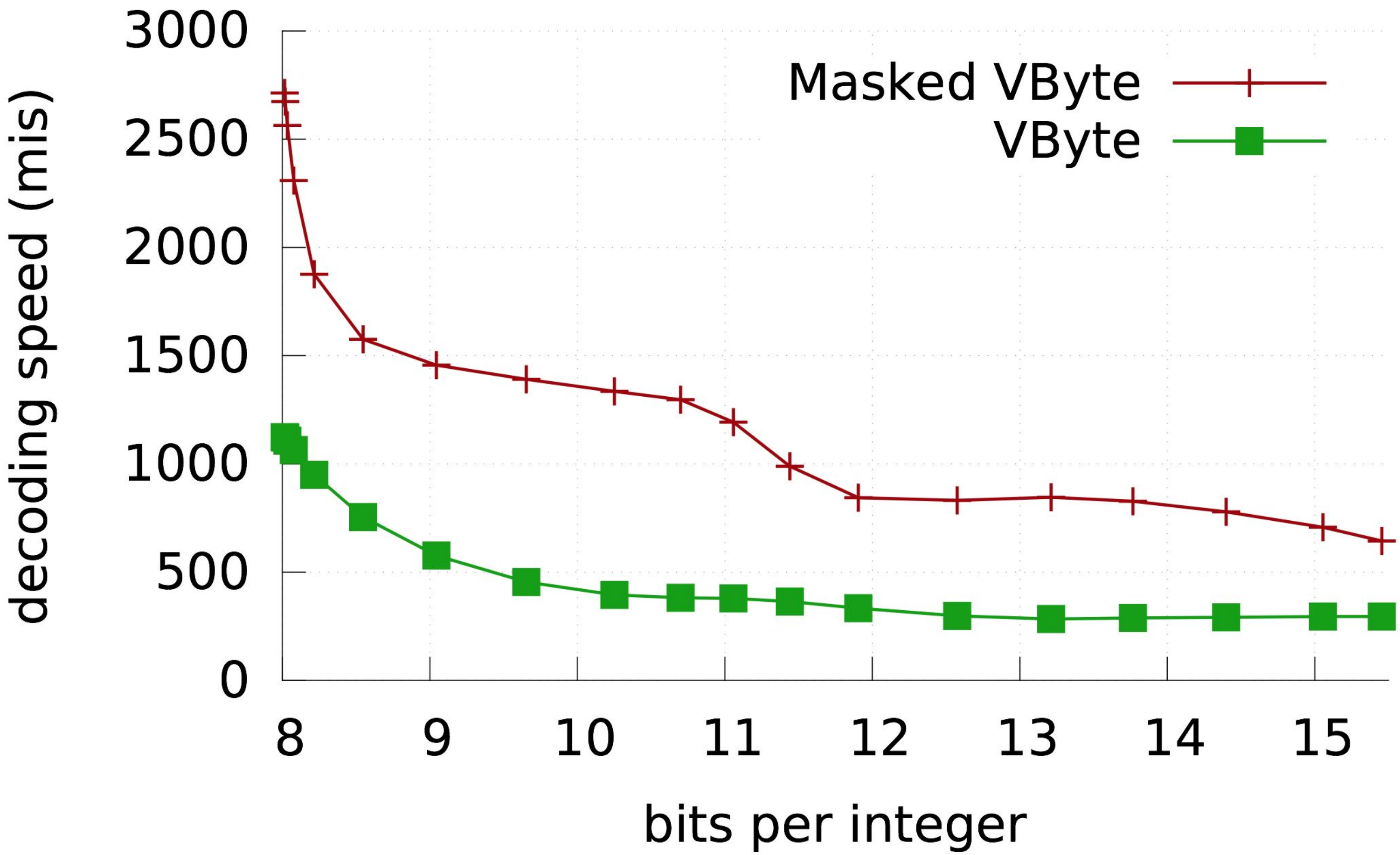
```
00111101 10111101  01001110 11101110
```

Final result!

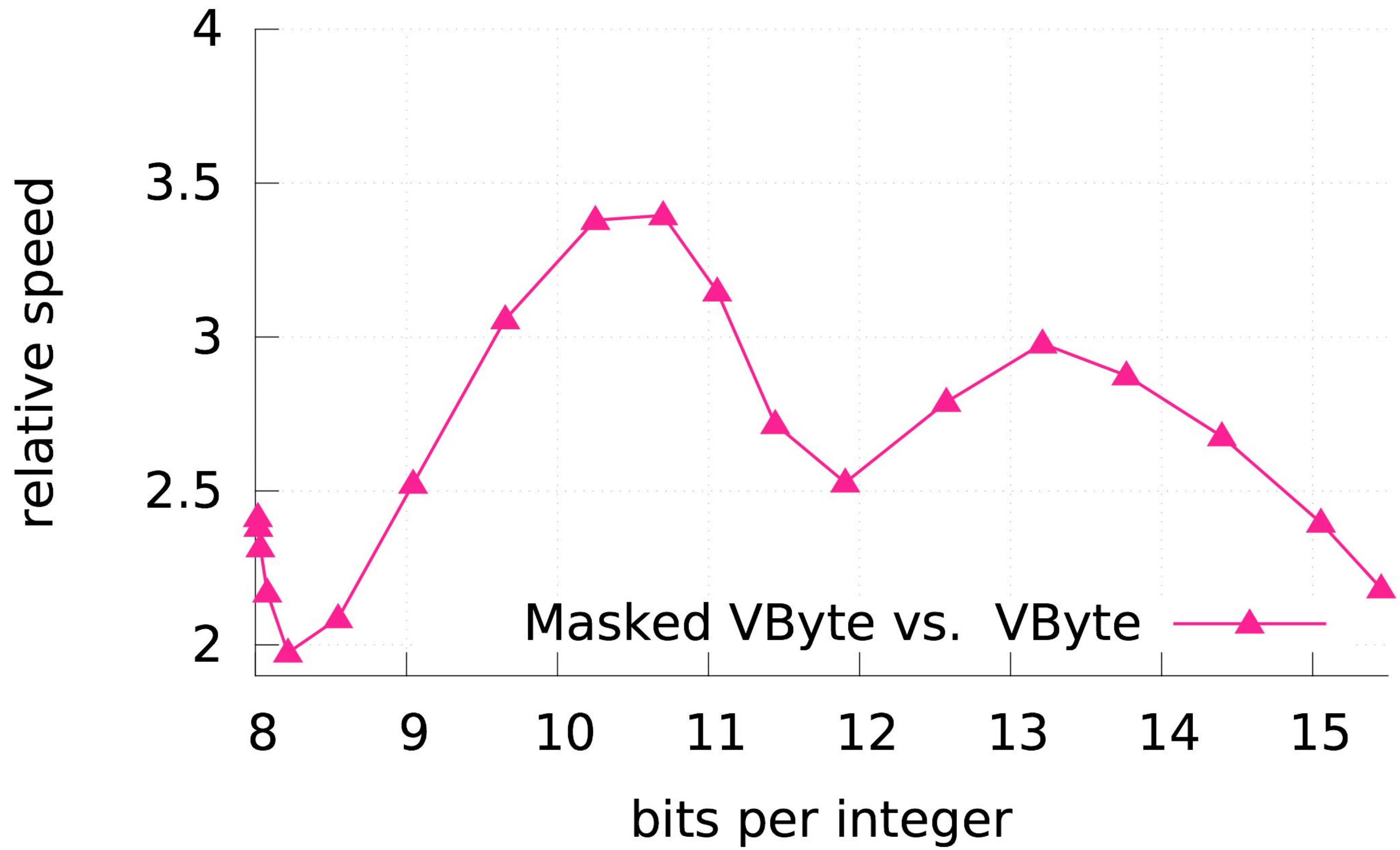
Checking my work against initial varint:

```
11101110 10011101  11110101 11101101
00000011
```

Results


decoding speed (mis)
3000
2500
2000
1500
1000
500
0
Masked VByte
VByte
8
9
10
11
12
13
14
15
bits per integer

Results


4
3.5
3
2.5
2
relative speed
8
9
10
11
12
13
14
15
bits per integer
Masked VByte vs. VByte

Q&A