Wu *Algorithms Mol Biol (2016) 11:5*
DOI 10.1186/s13015-016-0069-5

Algorithms for Molecular Biology

RESEARCH

Open Access

Bitpacking techniques for indexing genomes: I. Hash tables

Thomas D. Wu*

Abstract

Background: Hash tables constitute a widely used data structure for indexing genomes that provides a list of genomic positions for each possible oligomer of a given size. The offset array in a hash table grows exponentially with the oligomer size and precludes the use of larger oligomers that could facilitate rapid alignment of sequences to a genome.

Results: We propose to compress the offset array using vectorized bitpacking. We introduce an algorithm and data structure called BP64-columnar that achieves fast random access in arrays of monotonically nondecreasing integers. Experimental results based on hash tables for the fly, chicken, and human genomes show that BP64-columnar is 3 to 4 times faster than publicly available implementations of universal coding schemes, such as Elias gamma, Elias delta, and Fibonacci compression. Furthermore, among vectorized bitpacking schemes, our BP64-columnar format yields retrieval times that are faster than the fastest known bitpacking format by a factor of 3 for retrieving a single value, and a factor of 2 for retrieving two adjacent values.

Conclusions: Our BP64-columnar scheme enables compression of genomic hash tables with fast retrieval. It also has potential applications to other domains requiring differential coding with random access.

Keywords: Hash table, Sequence alignment, Genomics, Data compression

Background

In bioinformatics applications, genomes are generally represented not as a linear string of nucleotides, but as indexed data structures that can facilitate various analyses, such as the rapid alignment of query reads. With the advent of high-throughput sequencing and the generation of unprecedented volumes of read data [1], speed has become paramount in genomic alignment. One major indexing method of preprocessing genomes for efficient alignment is a hash table. A hash table represents a genome sequence as multiple lists of genomic positions, one list for each possible *k*-mer, or oligomer of some preselected length *k*.

Hash tables are used by various programs, including BLAST [2], PATTERNHUNTER [3], SHRIMP [4], BLAT [5], NextGenMap [6], and GMAP [7], to identify short oligomer (or seed) matches between a read and the genome. These seeds can then be combined or extended to obtain a more complete alignment. Hash tables are particularly useful for aligning reads that include multiple mismatches or indels relative to a genome. Hash tables are also useful for applications where ambiguity arises in the nucleotide content at a given position, such as with single nucleotide polymorphisms, or SNPs, since a hash table can map two different oligomers onto the same position.

In genomics, hash tables are typically implemented as a simple lookup table [8], in which an offset array contains pointers into a positions array, for the universe of possible *k*-mers (Fig. 1a). This straightforward table implementation is feasible because of the fixed and relatively small value of *k* needed for our domain. More general domains, having keys of arbitrary or relatively long length, require a hash function to compute a bucket index. Hash functions raise the possibility of collisions, where different keys map to the same bucket index, which then necessitate potentially complex and time-consuming procedures for handling such collisions.

*Correspondence: twu@gene.com
Department of Bioinformatics and Computational Biology, Genentech, Inc., 1 DNA Way, 94080, South San Francisco, CA 94080, USA

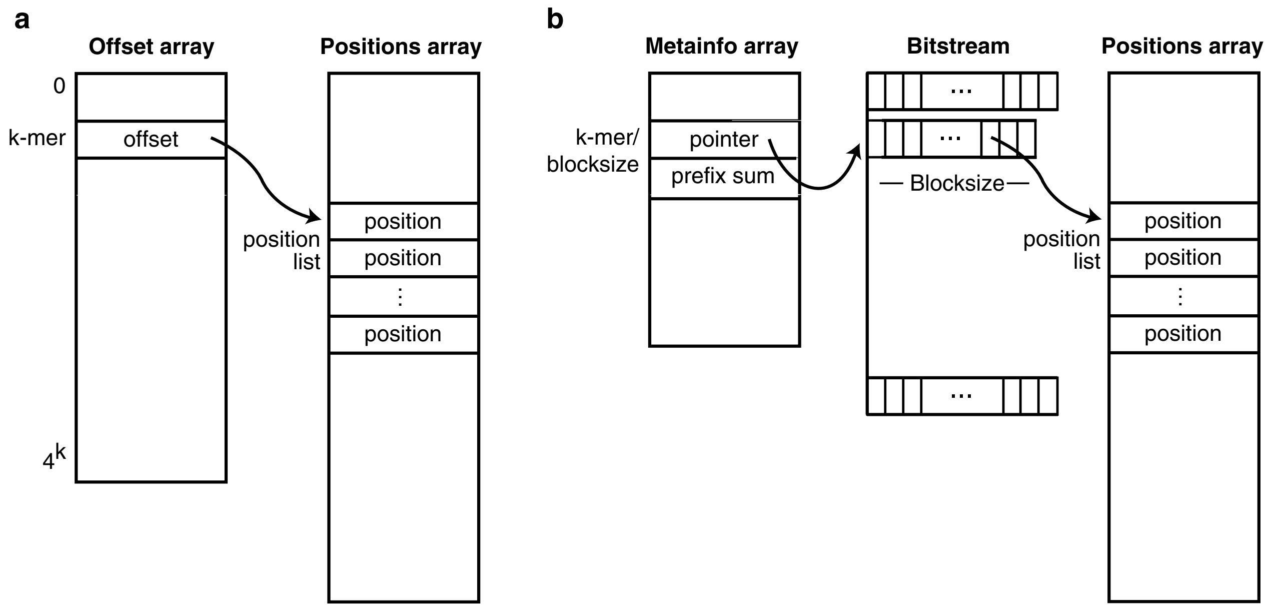


Fig. 1 Hash table representation. **a** Standard representation using an offset array that indicates the start of genomic positions for a given *k*-mer. **b** Compressed representation where the offset array has been replaced by a bitstream and a metainformation array. The bitstream contains differences between offsets that have been compressed by bitpacking them into blocks of a given size. The metainformation array contains a pointer and a prefix sum for every block

In contrast, a lookup hash table directly provides a list of positions in the genome containing a given *k*-mer. Obtaining that list requires extracting two adjacent values from the offset array: one representing the starting pointer in the positions file for the given *k*-mer, and one for the following pointer (i.e., for the *k*-mer incremented by 1). The difference between these two offsets represents the number of entries in the positions array for the given *k*-mer.

One limitation of lookup hash tables is that the size of the offset array grows exponentially with *k*. Consequently, existing applications have generally used small *k*-mer sizes, such as 11-mers for BLASTN and BLAT, and 12-mers for our initial implementation of GMAP. However, larger oligomers are generally preferable for performing search, because their greater specificity can greatly reduce the number of candidate genomic positions that need to be processed. Accordingly, the developers of one hash table-based alignment program FusionMap found that using 14-mers gave a speedup of 2–4 times over using 12-mers for alignment [9].

Although modern computers can access increasingly large amounts of primary random-access memory, or RAM, storage space still remains an issue for storing and using large data structures. As the authors of the PatternHunter II program stated, their development was hampered by the the "large memory requirements for multiple hashtables" [10]. Memory considerations also likely precluded the developers of FusionMap from extending their idea beyond 14-mers, since a hash table for 15-mers would require the offset array to be 4 GB in size.

In this paper, we address this problem by using compression. Compression allows larger amounts of information to be stored within a limited amount of memory. Our proposed compression scheme depends on the observation that values in the offset array are monotonically non-decreasing, because they point to successive locations in the positions array. In other words, each value in the offset array is equal to or greater than the previous one. In this situation, compression can be applied to the differences between adjacent values; such differences are generally small integers, which can be compressed more efficiently than large integers. This type of compression has been termed *differential coding*, as opposed to *direct coding*, where successive values may increase or decrease in value, but the original values themselves tend to be small.

Specifically, we explore an approach to compressing offset arrays using a technique called *vectorized bitpacking*, in which blocks of integers are represented with fewer than the 32 bits that are normally allocated for them, and then accessed and processed in parallel. Various schemes for vectorized bitpacking have been developed, but they have been designed for the problem of decoding streams of integers serially. In particular, a recently published scheme for vectorized bitpacking called BP128 [11] is able to achieve very high throughputs for serial decoding, at a rate of 2 billion integers per second, which represents the fastest known method to date.

However, for the genomic problem of compressing and decoding offset arrays in hash tables, we do not require decoding of all integers serially, but rather decoding of individual integers with *random access*. In particular, for a given *k*-mer, we wish to decode two adjacent elements from an arbitrary location within a compressed offset array. In this paper, we introduce a vectorized bitpacking scheme that is relatively compact and extremely fast for this task, and we demonstrate its effectiveness for genomic applications.

Methods

Vectorized bitpacking

As its name suggests, vectorized bitpacking involves two concepts: bitpacking and vectorization. In bitpacking, we attempt to conserve computer storage space by using as few bits as possible for storing each integer. Normally, arrays of integers are represented by using a single 32-bit word for each integer, which can represent positive values from 0 to $2^{32} - 1$, or approximately 4 billion. However, in many applications, including ours, integer values (or their differences) have small values that do not need all 32 bits in each word. A bitpacking scheme, therefore, uses fewer than 32 bits for each integer when possible.

Many bitpacking schemes are already well established for compressing integers. For example, one example of bitpacking is a universal code, such as Elias gamma coding [12]. In Elias gamma coding, a positive integer needing c bits can be encoded using $(2c - 1)$ bits, where the first $(c - 1)$ bits are zeroes, used as a codeword to indicate that the following c bits contain the actual integer. These leading $(c - 1)$ bits are needed in serial decoding to delineate the beginning and end of each integer. Likewise, a similar scheme called Elias delta coding [12] represents an integer by encoding its number of bits c with Elias gamma, and appending the $(c - 1)$ bits that follow the most significant bit. Another universal code uses sets of Fibonacci numbers to encode an integer [13], and is intended to help with decoding in the presence of noise, because it can recover from a damaged bitstream.

The idea behind the second concept, vectorization, is to design algorithms so that they access and process integers in parallel, rather than individually. One type of vectorization makes use of special computer instructions for processing integers in parallel. These SIMD (single instruction, multiple data) instructions exploit specialized 128-bit and 256-bit registers that have been incorporated into processors over the past decade, and which are scheduled to expand to 512-bit registers in the near future. For example, a 128-bit register allows a processor to process four 32-bit integers in parallel, with SIMD operations such as shifting all four integers rightward by a certain number of bits, or adding the four integers in one register to the four integers in another register simultaneously.

Vector-based operations require that integers be encoded with a uniform bit length. Therefore, unlike universal codes, where each integer can have a distinct number of bits, a vectorized bitpacking scheme must allocate the same number of bits for the integers. This does not mean that a single bit width needs to hold for the entire array of integers. Rather, an array can be divided into blocks of a predetermined size, such as 128 or 64 integers at a time, and all integers within each block can then be represented using a uniform number of bits. The bit width for each block is selected to be sufficient for the largest value in that block.

The bit width for each block represents an attribute that needs to be stored in a separate data structure we call the *metainformation* array (Fig. 1b). For random-access applications, the metainformation array also needs to contain, for each block, a pointer into the bitstream where that block begins. If the bit width can be inferred from the total size of the block, then the metainformation array can represent the bit width implicitly based on the difference between successive pointer values.

Differential coding, which computes differences between adjacent integers before encoding, requires computation of a cumulative sum after decoding to obtain the original ascending values. This value is called a *prefix sum*. For serial applications, a prefix sum can be computed as a running total from the beginning of the data stream. However, for random-access applications, it would be computationally expensive to repeatedly compute each prefix sum from the beginning of the data stream. Therefore, computational efficiency can be achieved by storing intermediate prefix sums, one for the start of each block, so that the cumulative sum for any position can be obtained by summing only a relatively few difference values starting from the preceding intermediate prefix sum.

Layouts for vectorized bitpacking

When the bit width for a block is less than 32, more than one integer can be stored in each 32-bit word. The way in which integers are arranged in words can be considered a *layout*. Two layouts have been proposed so far for vectorized bitpacking. In a horizontal layout, adjacent integers are packed next to each other [14]. Figure 2a shows the horizontal layout for a bit width of 6, where each row represents a 128-bit vector of four 32-bit words that are accessed in parallel. Integers that are processed in parallel

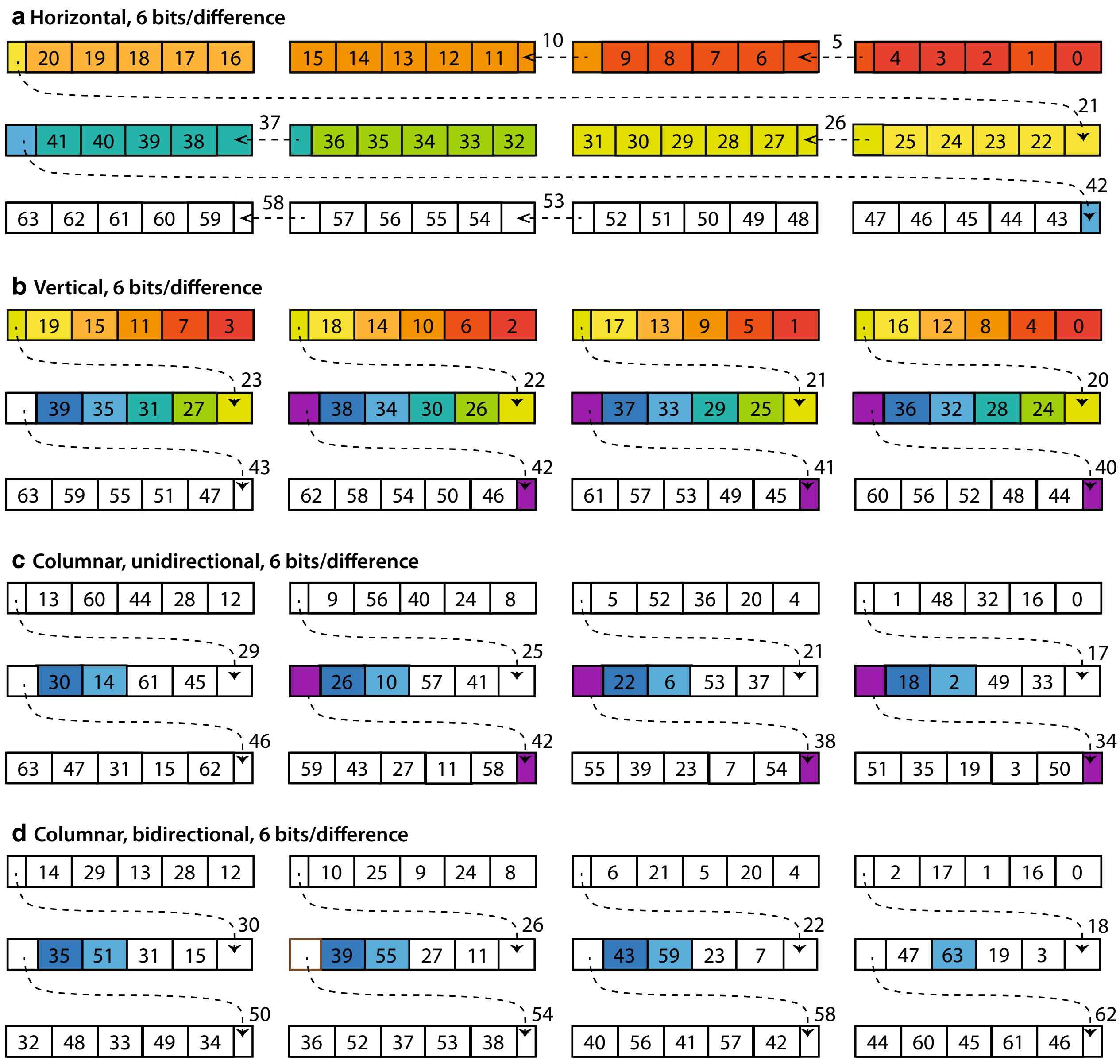
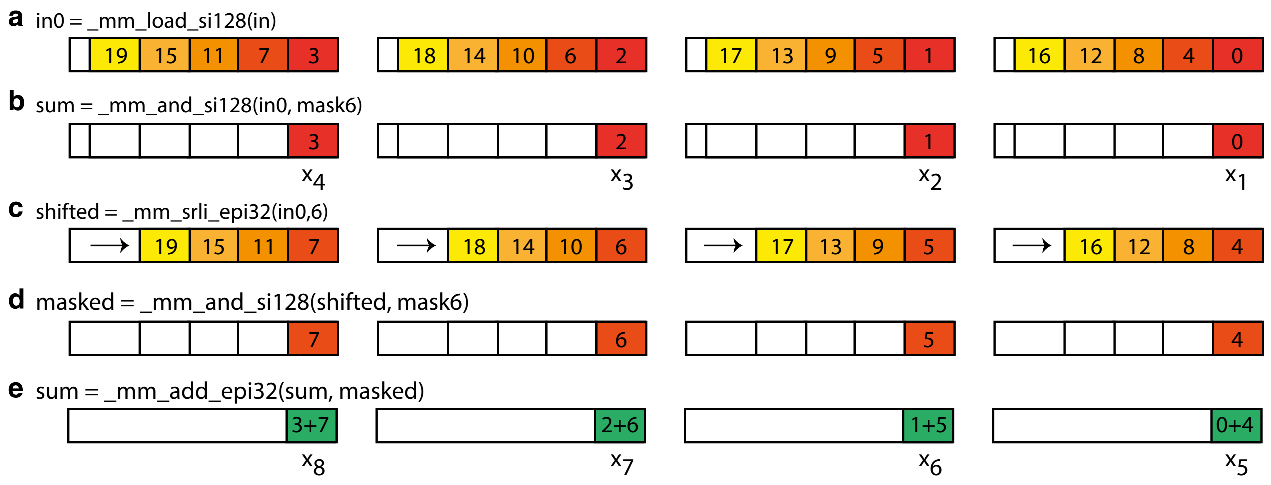

a Horizontal, 6 bits/difference
b Vertical, 6 bits/difference
c Columnar, unidirectional, 6 bits/difference
d Columnar, bidirectional, 6 bits/difference

Fig. 2 Packing layouts for block-based differential coding. Each layout encodes a block of 64 difference values using 6 bits for each value. Values are placed into three 128-bit registers, each containing four 32-bit words. Each *small box* shows a 6-bit quantity, labeled with an integer *r* that indicates the place for storing the difference value d_r. Some 6-bit quantities are spread over two words, as shown by *dashed lines*. Blocks of *color* indicate parallel processing of the difference values needed to decode the value x_{43}. **a** *Horizontal* layout, with values stored in index order. **b** *Vertical* layout, with values striped across each set of words, as used in BP128. **c** Columnar layout, unidirectional scheme, with indices at increments of 4 striped across each set of words. **d** Columnar layout, bidirectional scheme

are shaded with the same color. In the decoding process, this layout requires a separate shift operation for each integer in the word. Although SIMD operations exist for performing distinct shifts for the words in a register, they entail a fair amount of complexity in the decoding algorithm.

In contrast, the vertical layout used in BP128 can be processed by SIMD operations in a more straightforward manner. This layout stripes integers in sets of four across each set of four words [11, 15, 16]. Figure 2b shows a vertical layout for a bit width of 6. Decoding in this layout loads four 32-bit words in parallel (Fig. 3a) and masks their low-order bits to obtain the first four values (Fig. 3b). Then, the next four values can be obtained by right-shifting the words in parallel by the uniform bit width for the block (Fig. 3c), and again masking their


a in0 = _mm_load_si128(in)
19 15 11 7 3
18 14 10 6 2
17 13 9 5 1
16 12 8 4 0
b sum = _mm_and_si128(in0, mask6)
3 2 1 0
x4 x3 x2 x1
c shifted = _mm_srli_epi32(in0,6)
19 15 11 7
18 14 10 6
17 13 9 5
16 12 8 4
d masked = _mm_and_si128(shifted, mask6)
7 6 5 4
e sum = _mm_add_epi32(sum, masked)
3+7 2+6 1+5 0+4
x8 x7 x6 x5
f

```
const static __m128i mask6 = _mm_set1_epi32(63U);

void unpack_06 (__m128i *out, __m128i *in) {
  __m128i in0 = _mm_load_si128(in);  // a
  __m128i sum = _mm_and_si128(in0, mask6);  // b
  _mm_store_si128(out++, sum);

  __m128i shifted = _mm_srli_epi32(in0,6);  // c
  __m128i masked = _mm_and_si128(shifted, mask6);  // d
  sum = _mm_add_epi32(sum, masked);  // e
  _mm_store_si128(out++, sum);

  ... [continue shift/mask/add to decode entire block] ...
}
```

Fig. 3 Decoding of vertical layout. An example is shown for the first two cycles of serial decoding of the vertical layout from a block packed with a bit width of 6. *Shaded* regions correspond to the values in Fig. 2b. Source code is shown in part (**f**), with key steps shown graphically in parts (**a**) through (**e**). **a** Loading of the first 128-bit vector from the block. **b** Masking of the first four difference values from the vector. **c**, **d** Shifting and masking of the second four difference values from the vector. **e** Parallel addition of the first and second vectors of difference values. **f** Source code in the C language. *Comments* in the source code correspond to the steps labeled **a** through **e**

low-order bits (Fig. 3d). Since these values represent the differences between the original values, we use a SIMD operation to keep a running sum of the shifted-and-masked low-order bits (Fig. 3e). The remaining values in the block can be decoded using the same SIMD shift and mask steps, with a new set of four 32-bit words loaded whenever all of the currently loaded difference values have been processed. Source code for decoding a vertical layout is shown in Fig. 3f.

In the BP128 algorithm, because four cumulative sums are maintained in parallel, it is not the differences between adjacent values (i.e., $x_{i+1} - x_i$) that we want for differential coding, but rather the differences between each value and the one four elements away (i.e., $x_{i+4} - x_i$). An exception holds for the first four differences in each block, which are computed for x_1, x_2, x_3, and x_4 relative to the prefix sum x_0 for the beginning of the block, which acts as the starting point for computing the four cumulative sums. The overall cyclic difference scheme for a block of 64 is shown in Fig. 4a.

Bitpacking for fast random data access

Computing cumulative sums in BP128 is expensive for random-access applications, since the decoding procedure is designed to extract the entire block of 128 integers. In other words, BP128 performs computations that are irrelevant if we want only a single cumulative sum from a given block. To improve vectorized bitpacking for random-access applications, we introduce the following modifications: (1) reduce the block size, (2) arrange integers within each block in a columnar layout, and (3) utilize a bidirectional scheme for computing differences between values. These modifications make it possible to decode a single value in a differentially encoded block with greater efficiency than the BP128 scheme. We discuss each of these modifications in turn.

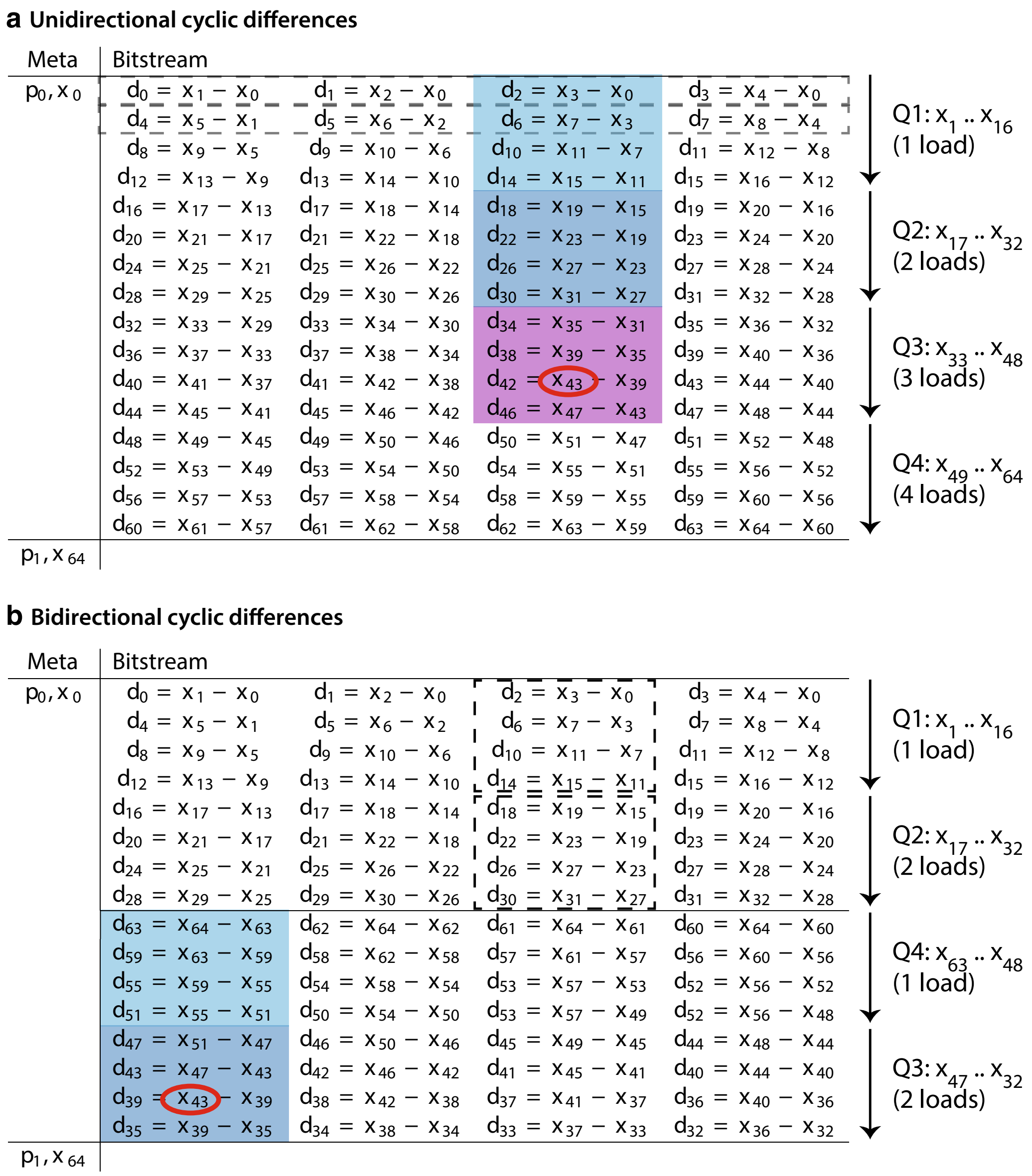

a Unidirectional cyclic differences
Meta
Bitstream
p0, x0
d0 = x1 − x0 | d1 = x2 − x0 | d2 = x3 − x0 | d3 = x4 − x0
d4 = x5 − x1 | d5 = x6 − x2 | d6 = x7 − x3 | d7 = x8 − x4
d8 = x9 − x5 | d9 = x10 − x6 | d10 = x11 − x7 | d11 = x12 − x8
d12 = x13 − x9 | d13 = x14 − x10 | d14 = x15 − x11 | d15 = x16 − x12
d16 = x17 − x13 | d17 = x18 − x14 | d18 = x19 − x15 | d19 = x20 − x16
d20 = x21 − x17 | d21 = x22 − x18 | d22 = x23 − x19 | d23 = x24 − x20
d24 = x25 − x21 | d25 = x26 − x22 | d26 = x27 − x23 | d27 = x28 − x24
d28 = x29 − x25 | d29 = x30 − x26 | d30 = x31 − x27 | d31 = x32 − x28
d32 = x33 − x29 | d33 = x34 − x30 | d34 = x35 − x31 | d35 = x36 − x32
d36 = x37 − x33 | d37 = x38 − x34 | d38 = x39 − x35 | d39 = x40 − x36
d40 = x41 − x37 | d41 = x42 − x38 | d42 = x43 − x39 | d43 = x44 − x40
d44 = x45 − x41 | d45 = x46 − x42 | d46 = x47 − x43 | d47 = x48 − x44
d48 = x49 − x45 | d49 = x50 − x46 | d50 = x51 − x47 | d51 = x52 − x48
d52 = x53 − x49 | d53 = x54 − x50 | d54 = x55 − x51 | d55 = x56 − x52
d56 = x57 − x53 | d57 = x58 − x54 | d58 = x59 − x55 | d59 = x60 − x56
d60 = x61 − x57 | d61 = x62 − x58 | d62 = x63 − x59 | d63 = x64 − x60
p1, x64
Q1: x1 .. x16 (1 load)
Q2: x17 .. x32 (2 loads)
Q3: x33 .. x48 (3 loads)
Q4: x49 .. x64 (4 loads)
b Bidirectional cyclic differences
Meta
Bitstream
p0, x0
d0 = x1 − x0 | d1 = x2 − x0 | d2 = x3 − x0 | d3 = x4 − x0
d4 = x5 − x1 | d5 = x6 − x2 | d6 = x7 − x3 | d7 = x8 − x4
d8 = x9 − x5 | d9 = x10 − x6 | d10 = x11 − x7 | d11 = x12 − x8
d12 = x13 − x9 | d13 = x14 − x10 | d14 = x15 − x11 | d15 = x16 − x12
d16 = x17 − x13 | d17 = x18 − x14 | d18 = x19 − x15 | d19 = x20 − x16
d20 = x21 − x17 | d21 = x22 − x18 | d22 = x23 − x19 | d23 = x24 − x20
d24 = x25 − x21 | d25 = x26 − x22 | d26 = x27 − x23 | d27 = x28 − x24
d28 = x29 − x25 | d29 = x30 − x26 | d30 = x31 − x27 | d31 = x32 − x28
d63 = x64 − x63 | d62 = x64 − x62 | d61 = x64 − x61 | d60 = x64 − x60
d59 = x63 − x59 | d58 = x62 − x58 | d57 = x61 − x57 | d56 = x60 − x56
d55 = x59 − x55 | d54 = x58 − x54 | d53 = x57 − x53 | d52 = x56 − x52
d51 = x55 − x51 | d50 = x54 − x50 | d53 = x57 − x49 | d52 = x56 − x48
d47 = x51 − x47 | d46 = x50 − x46 | d45 = x49 − x45 | d44 = x48 − x44
d43 = x47 − x43 | d42 = x46 − x42 | d41 = x45 − x41 | d40 = x44 − x40
d39 = x43 − x39 | d38 = x42 − x38 | d37 = x41 − x37 | d36 = x40 − x36
d35 = x39 − x35 | d34 = x38 − x34 | d33 = x37 − x33 | d32 = x36 − x32
p1, x64
Q1: x1 .. x16 (1 load)
Q2: x17 .. x32 (2 loads)
Q4: x63 .. x48 (1 load)
Q3: x47 .. x32 (2 loads)

Fig. 4 Difference schemes for vectorized differential coding. These schemes show the bitstream and the metainformation values for a given block, with p_0 and p_1 indicating the positions in the bitstream where this block and the next block begin, and x_0 and x_{64} indicating the prefix sum for the two blocks. These schemes show how the original ascending values x in a block can be converted to difference values d. **a** Unidirectional cyclic differences as used in BP128, except shown here for a block size of 64. The difference d_i involves the original values x_{i+1} and x_{i-3}. An exception holds for the *first row*, where differences are taken relative to x_0, the prefix sum for the block. *Dashed boxes* indicate the first two processing steps for the vertical layout. **b** Bidirectional cyclic difference scheme, which matches the unidirectional scheme for the first half of the block, but computes differences relative to x_{64} for the second half of the block. *Dashed boxes* indicate the first two processing steps for column 2. In both parts, *shaded regions* correspond to the values needed to compute the *circled values* x_{43}, requiring 3 loads for the unidirectional scheme and 2 loads for the bidirectional scheme. The *colors* correspond to those in Fig. 2c, d. *Q1–Q4* indicate quarter blocks, and are annotated with the total number of SIMD loads required

Reduced block size

In vectorized bitpacking, integers are encoded using a uniform bit width for each block. The BP128 scheme is so named because it packs integers in blocks of 128. One reason for using that block size is that encoding 128 integers, regardless of which bit width from 0 through 32 is used, results in a bitstream that is aligned on a 128-bit word boundary, which can improve the speed of SIMD load instructions on some computer systems [17].

For serial applications, since all values in a block are decoded, the choice of block size should have little effect on overall decoding speed. In fact, a larger block size may be slightly more efficient for exhaustive decoding, because each call to a decoding procedure for a single block can handle a larger amount of work.

However, for random-access applications, a smaller block size is potentially advantageous in reducing the amount of computation needed within a given block to obtain a single value. In other words, a block size of 64 should roughly halve the number of SIMD load, shift, mask, and addition operations needed to decode a single value, relative to a block size of 128. The actual effect on decoding speed may not achieve these theoretical savings, though, because of the time required for memory access and the effect of pipelining of operations by the processor. The drawback of halving the block size is that twice as many pieces of metainformation (pointers and block prefix sums) are required, thereby increasing the amount of space needed.

One advantage of a block size of 64 is that it allows us to restrict the uniform bit widths to even values. As we mentioned previously, the BP128 scheme aligns its block of 128 integers at a 128-bit word boundary, for every bit width from 0 through 32, without any need for zero padding. For example, for a bit width of 3, a block of 128 integers requires exactly three 128-bit registers for storage. However, for a block size of 64, odd bit widths do not offer any savings in storage relative to the next larger even bit width. For this block size, a bit width of 3 uses one and a half 128-bit registers, effectively the same space as for a bit width of 4, which uses two full 128-bit registers. Furthermore, the decoding procedures for odd bit widths are more complex than those for even bit widths, because odd sizes do not fit evenly into 32-bit registers, necessitating more SIMD operations for merging integers that cross from the end of one register into the beginning of the next one (as indicated by the dashed lines in Fig. 2). Consequently, decoding speeds for even bit widths have been shown to be slightly faster than for odd bit widths [16].

Although we will continue our exposition of our bit-packing scheme using a block size of 64, we will consider experimentally an even smaller block size of 32. Such a block size would allow us to restrict bit widths to multiples of 4, and should theoretically reduce the number of SIMD operations further, albeit at a cost of further doubling the storage needed for metainformation. A block size of 64 is somewhat more natural for genomic applications, since it is a power of 4, meaning that the beginning of each block can be associated with a specific *k*-mer. However, a natural interpretation is not essential for either serial or random-access decoding, so arbitrary block sizes can be considered, although some may turn out to be more or less optimal for compression or speed.

Columnar packing layout

We can greatly improve the efficiency of random access for differential decoding by noting that most of the SIMD computations in a BP128 decoding are wasted if we need only a single cumulative sum in the block. For example, consider the case where we require only the cumulative sum for the 43rd entry, or x_{43}. Fig. 2a, b show with color coding the loads needed for the horizontal and vertical layouts. In both layouts, all difference values from d_0 through d_{42} must be decoded and summed in order to obtain the value for x_{43}. For the vertical layout, a total of 11 sets of decoding steps are needed to reach x_{43}, with the first two decoding steps represented by dashed boxes in Fig. 4a.

However, the critical path for obtaining x_{43} can be shortened if we can load the difference values by columns instead, as shown by the light blue, dark blue, and violet shading in Fig. 4a. Then, the path to computing x_{43} involves summing the difference values in the columnar path from d_2 through d_{42}, and requires only three decoding steps instead of 11.

To decode difference values by columns instead of rows, we propose a new packing layout called a columnar layout (Fig. 2c). In this layout, we pack one column at a time, with the 16 entries in column 0 (d_0, d_4, ..., d_{60}) striped in groups of four across available spaces in a series of 128-bit vectors. The remainder of the block is similarly packed in column order for columns 1, 2, and 3. The SIMD loads needed for obtaining x_{43} are shaded in light blue, dark blue, and violet in Fig. 2c.

Bidirectional difference scheme

The columnar layout requires us to implement a separate decoding procedure for each column in each quarter-block, where the first quarter-block (Q1) contains block positions 1–16; Q2 contains 17–32; the Q3 contains 33–48; and Q4 contains 48–63 (Fig. 4a). Entries in Q1 need a single SIMD load of integers, whereas entries in Q2, Q3, and Q4 need two, three, and four SIMD loads, respectively.

In this scheme, the Q3 and Q4 entries require more loads, since those values are farther away from the prefix sum x_0, which constitutes the starting point for computing cumulative sums. However, we can improve the situation if we consider that each block has both a beginning prefix sum x_0 and an ending prefix sum x_{64}, which is already stored in the metainformation array as the beginning prefix sum for the next block. Therefore, we can compute the differences for the first half-block (values x_0 through x_{32}) relative to the beginning prefix sum, and the differences for the second half-block (values x_{32} through x_{64}) relative to the ending prefix sum. (The sum x_{32} can be computed from either the beginning or the ending prefix sum.) Hence, to compute a value for the second half of the block, we can perform SIMD operations just on that half, and then subtract the resulting cumulative sum from x_{64}.

This idea gives rise to a bidirectional difference scheme (Fig. 4b), rather than the unidirectional scheme (Fig. 4a) that we have been considering so far. Consequently, the bidirectional scheme allows for fewer loads of data for quarter blocks Q3 and Q4, which now require only two and one SIMD loads, respectively. For any given entry x_r at block position r, we can compute the desired column by first computing the distance $\delta = 31 - |r - 32|$ to the nearest prefix sum, which is equivalent to $\delta = r - 1$ for $r \leq 32$ and $\delta = 63 - r$ for $r \geq 32$. Then we require only the values in column (δ mod 4), beginning from row 0 in the half-block to row $\lfloor \delta/4 \rfloor$.

For our example of computing x_{43} in Q3, the desired column in the bidirectional scheme is shaded in Fig. 4b as light blue and dark blue, requiring two SIMD loads, as opposed to three. These shades correspond to the bidirectional columnar layout shown in Fig. 2d. In this layout, we pack the integers in the first half-block according to the columns in Fig. 4b, and then the integers in the second half-block, again according to columns, but in reverse order.

Within-column summation

To decode a value from the bitcompressed differences, we need a total of 256 specialized procedures, to handle all combinations of the 16 possible even bit widths, with four columns for each of the four quarter-blocks. A jump table for x_r can be used to invoke the appropriate decoding procedure, based on (1) the bit width, (2) the column (δ mod 4), and (3) the quarter-block, based on $\lfloor r/16 \rfloor$. A bit width of 0 indicates the special case in which all values in the block are zero; for this case, the decoding procedure can simply return the prefix sum x_0 for any entry in the block.

As an example of one of these 256 procedures, Fig. 5 shows a procedure for decoding column 2 for the Q2 quarter block from a block with a bit width of 6. The two decoding steps are also shown by the dashed boxes in Fig. 4b. The first decoding step yields four difference values, shown as words 0 through 3 in Fig. 5c, and the second decoding step yields another four difference values, shown in Fig. 5d. Our procedure then performs a final SIMD addition to add the first set of difference values to the second set, thereby yielding words 4 through 7 in Fig. 5e. Words 0 through 7 from Fig. 5c, e can be stored in an array, as shown in Fig. 5g.

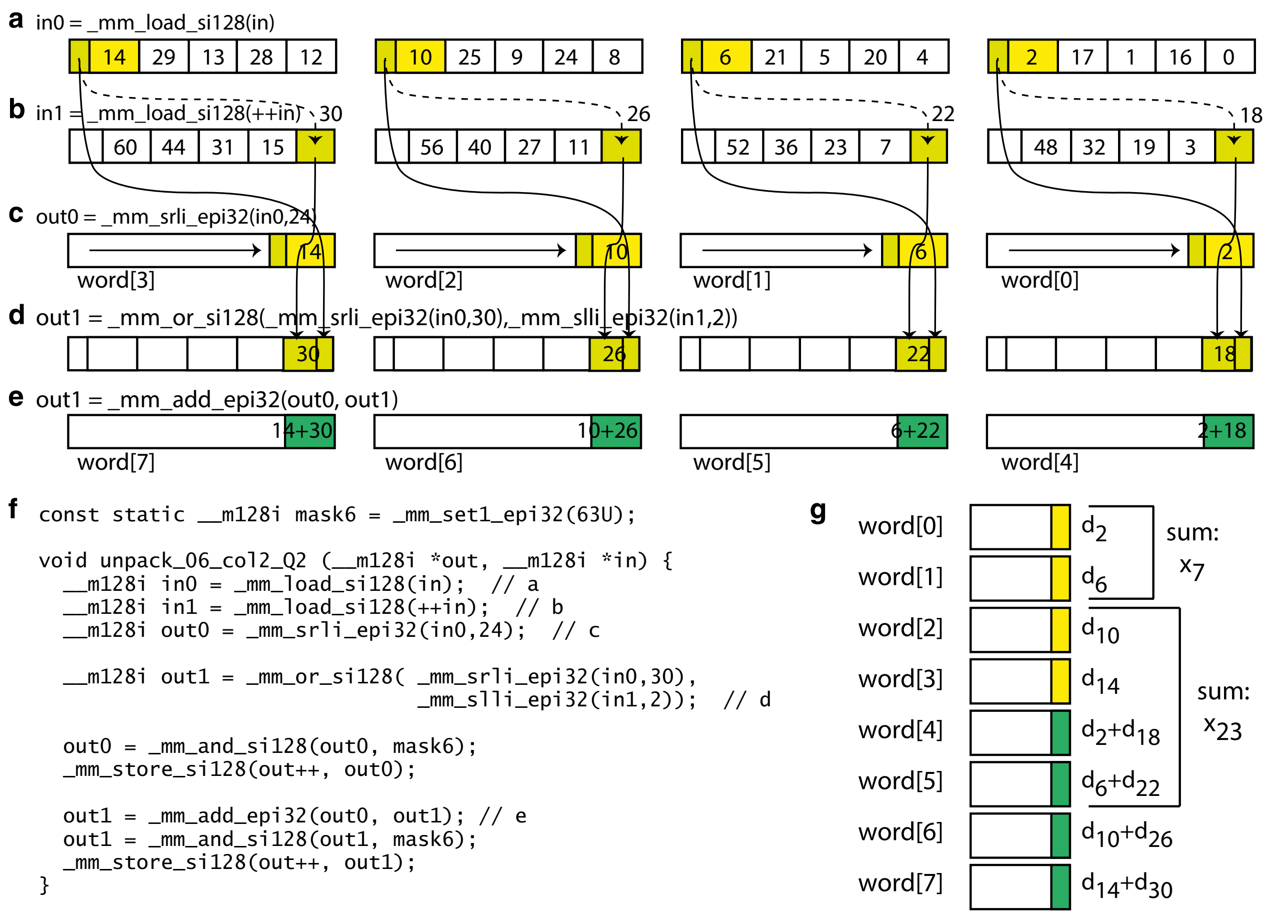
Through appropriate summations of the first four words, we can obtain the cumulative sums x_3, x_7, x_{11}, and x_{15} for Q1, by using 1, 2, 3, and 4 terms, respectively. It would therefore seem, by extension, that obtaining the cumulative sums x_{19}, x_{23}, x_{27}, and x_{31} for Q2 would require computing over 5, 6, 7, and 8 terms, respectively. However, our final SIMD computation is designed to perform some additions in parallel, and therefore makes summation of at most four terms sufficient in all cases. For example, to compute x_{23}, we need add only words 2 through 5, which represent the four quantities d_{10}, d_{14}, $(d_2 + d_{18})$, and $(d_6 + d_{22})$ instead of the expected addition of six separate quantities d_2, d_6, d_{10}, d_{14}, d_{18}, and d_{22}.

From this example, we also see that the entries for Q1 are decoded on the way to decoding the Q2 entries. Likewise, the entries for Q4 are decoded on the way to decoding the Q3 entries. Therefore, we need to devise only 128 distinct procedures for decoding the Q2 and Q3 quarter-blocks, with the Q1 and Q4 procedures being derived easily as the initial parts of those procedures.

Retrieval of two adjacent offset values

Although some applications of differential coding require retrieval of only a single value, our domain of interest, genomic alignment, requires us to decode two adjacent offsets. These two values specify the endpoints for the list of genomic positions for a given k-mer, representing the start and end of the list in the positions array. In other words, when we decode x_{23} to obtain the start of the list, we must also decode x_{24} to obtain the end of the list. For a vertical layout, the additional value requires just one more difference value to be extracted and added. However, since our bitpacking scheme is designed to extract columns of values, it requires an entire second decoding procedure, in this case to extract column 3 of the Q2 entries. The most straightforward implementation to retrieve two adjacent offset values would be to make separate calls to two decoding procedures, which we call a *two-pass* implementation.

However, a slight improvement can be made by noting that in our difference scheme, adjacent values will be in neighboring columns (where we consider column 3 and column 0 to be neighboring). Therefore, decoding of column 0 will always be followed by column 1; column 1 by column 2; column 2 by column 3; and column 3 by column 0. Because adjacent columns are packed in adjacent stripes in the columnar layout, difference values from the second column may have already been loaded during computations for the first column. We can therefore combine the two separate decoding procedures to yield a *one-pass* implementation. A one-pass implementation can achieve some efficiency savings by avoiding duplicate SIMD loads of registers. These one-pass procedures can be derived by appropriate merging of individual two-pass procedures, taking advantage of registers that have already been loaded.


a in0 = _mm_load_si128(in)
14 29 13 28 12
10 25 9 24 8
6 21 5 20 4
2 17 1 16 0
b in1 = _mm_load_si128(++in)
30 26 22 18
60 44 31 15
56 40 27 11
52 36 23 7
48 32 19 3
c out0 = _mm_srli_epi32(in0,24)
14 10 6 2
word[3] word[2] word[1] word[0]
d out1 = _mm_or_si128(_mm_srli_epi32(in0,30),_mm_slli_epi32(in1,2))
30 26 22 18
e out1 = _mm_add_epi32(out0, out1)
14+30 10+26 6+22 2+18
word[7] word[6] word[5] word[4]
f
g
word[0] word[1] word[2] word[3] word[4] word[5] word[6] word[7]
d2 d6 d10 d14 d2+d18 d6+d22 d10+d26 d14+d30
sum: x7
sum: x23

```
const static __m128i mask6 = _mm_set1_epi32(63U);

void unpack_06_col2_Q2 (__m128i *out, __m128i *in) {
  __m128i in0 = _mm_load_si128(in);  // a
  __m128i in1 = _mm_load_si128(++in);  // b
  __m128i out0 = _mm_srli_epi32(in0,24);  // c

  __m128i out1 = _mm_or_si128( _mm_srli_epi32(in0,30),
                               _mm_slli_epi32(in1,2));  // d

  out0 = _mm_and_si128(out0, mask6);
  _mm_store_si128(out++, out0);

  out1 = _mm_add_epi32(out0, out1); // e
  out1 = _mm_and_si128(out1, mask6);
  _mm_store_si128(out++, out1);
}
```

Fig. 5 Decoding of columnar layout. An example is shown for decoding column 2 for quarter block Q2 from a block packed with a bit width of 6. Source code is listed in part (**f**), with key steps shown graphically in parts (**a**) through (**e**), and a final summation step in part (**g**). **a**, **b** Loading of two 128-bit vectors from the block. **c** Parallel (SIMD) right shift of each 32-bit word by 24 bits, to move d_2, d_6, d_{10}, and d_{14} into the lowest 6 bits. **d** Recombining of d_{18}, d_{22}, d_{26}, and d_{30}, which are split between two 128-bit vectors in the block, using a parallel right shift of the first vector by 30 bits and a parallel left shift of the second vector by 2 bits. **e** Parallel addition of the first and second vector of differences. **f** Source code in the C language. *Comments* in the source correspond to the steps labeled **a** through **e**. **g** Difference results shown as an array of 32-bit words. The value x_7 can be obtained by adding two terms from the array, while the value x_{23} can be obtained by adding four terms

Evaluation

Experimental setup

We evaluated the tradeoff between space and time for various methods to retrieve offsets from a hash table, both for a single offset and a pair of adjacent offsets. We used genomes of different sizes, namely, the fly genome (D. melanogaster version 5.25.64), chicken genome (Gallus gallus version 4), and human genome (version hg19).

We implemented our methods within the Succinct Data Structure Library (SDSL) 2.0 package, which is publicly available as C++ source code [18]. Our bit-packing code derives from a revision [16] of the original BP128 work [11], which we modified to implement our columnar method. To be consistent with the naming of classes in the SDSL package, our classes are named `bp64_encv_vector` for the BP64-vertical layout; `bp64_encc_vector` for BP64-columnar; and `bp32_encv_vector` for BP32-columnar. We also implemented benchmarking procedures to compare our methods with existing compression methods in SDSL, namely, Elias gamma, Elias delta, and Fibonacci encoding, all with a block size of 64, as well as with an uncompressed `int_vector` method available in SDSL. We added versions of these compression methods that encode the incremental value of 0 efficiently, by subtracting 1 from each positive integer during encoding and adding an equivalent amount when decoding an integer within a block. These modifications were implemented with the assistance of the author of SDSL (personal communication). Without these modifications, execution times of the original Elias and Fibonacci methods were extremely slow in our early benchmarking experiments.

All timing experiments were performed on a reserved Linux computer having 32 Intel Xeon E5-2667 v3 8-core processors running at 3.20 GHz. The computer had total memory of 264 GB and cache memory of 20 MB. The SDSL 2.0 library was compiled with the GNU g++ compiler, version 4.9.0, with the default settings, which turned off debugging code, and added the compiler flags "`-O3 -ffast-math -funroll-loops -msse4.2`".

Our benchmarking code generates 10 million random values uniformly over the space of possible queries and measures the average time to retrieve results for each query. Data structures were either read into memory from the filesystem or generated de novo in memory from the input files. A checksum was computed over the results to ensure that the methods gave consistent results and that the compiler did not optimize out the query. All timing measurements were repeated for 9 trials, with each trial involving different random values generated and testing different compression strategies in a randomly selected order. Results are summarized by the median over the 9 trials. We also measured the time for iterating through the 10 million queries, obtaining the test offset index, and performing the checksum, and subtracted the median times from all runs. These times amounted to a negligible fraction of the overall running times.

Source code for all bitpacking implementations of hash tables (and our companion research on bitpacking for enhanced suffix arrays) is made available as Additional file 1. The package is a modification of SDSL 2.0 that includes our new methods, benchmarking code, and alteration of existing methods to encode the incremental value of 0 efficiently. Genomic input files for the benchmarking experiments are hosted on a public Web site, with downloading instructions available within the package. Alternatively, we have prepared a package, available for download as Additional file 2, that allows users to generate their own benchmarks from any DNA or RNA source.

Retrieving a single offset value

We generated genomic hash tables using 15-mers, sampled every 3 bp in the genome. We used the offset array from each hash table as a source of monotonically nondecreasing values to be compressed in the following ways: (1) uncompressed, using the SDSL `int_vector` method, in which each offset was represented as a 4-byte quantity; (2) BP64-vertical, which is identical to the BP128 format as proposed by [11] that uses a unidirectional difference scheme and vertical cyclic packing format, but with a block size of 64 values and using only even-valued bit widths; (3) BP64-columnar, as proposed in this paper, using a bidirectional difference scheme and columnar packing format; (4) BP32-columnar, also proposed here, with a block size of 32; (5) Elias gamma encoding, with a block size of 64 values, as implemented in SDSL; (6) Elias delta encoding, with a block size of 64 values, as implemented in SDSL; and (7) Fibonacci encoding, with a block size of 64 values, as implemented in SDSL.

The results for retrieving a single offset value are shown in Fig. 6a. These results show that retrieval time is largely independent of genome size, which derives from the fact that the offsets file length depends instead on the *k*-mer size. The uncompressed format gives the fastest times at 12 ns/query, but requires 4 GB of space (4^{15} entries with 4 bytes required per entry). The Elias gamma, Elias delta, and Fibonacci formats produce compact representations that are 6–11 % of the uncompressed format, but generally have the slowest retrieval times, with over 130 ns/query for the fly genome, over 200 ns/query for the chicken genome, and 228–237 ns/query for human genomes. The BP64-vertical format requires slightly more space, at 8–14 % of the uncompressed format, but is 1.3–1.4 times faster than the SDSL methods, except for the fly genome, where the Elias gamma and delta methods are 10 % faster than the BP64-vertical method. The BP64-columnar format requires the same amount of space as BP64-vertical, but has retrieval times that are 2.7–3.0 times faster. The BP32-columnar format requires the most space among the bitpacking routines we tested, at 13–19 % of the uncompressed format (or 38–60 % more space than BP64-columnar), with times that are 12–15 % faster than BP64-columnar.

Retrieving two offset values

The results for retrieving two adjacent values are shown in Fig. 6b. The space measurements remain the same for each format relative to the first experiment, and only the time measurements differ. For the uncompressed, Elias gamma, Elias delta, and Fibonacci formats, we made two separate calls to the retrieval function. When we compare the results of Fig. 6b with those of Fig. 6a, we observe that times for two calls for the uncompressed data and SDSL methods were 1.2–1.7 times those for a single call. The fact that the retrieval time did not double reflects the effects of memory caching. For the BP64-vertical format, two separate calls to retrieve adjacent values required 1.1x of the time for a single call for one value, whereas for the BP64-columnar and BP32-columnar formats, the extra time required was a factor of 1.7-1.8. The difference between the vertical and columnar results

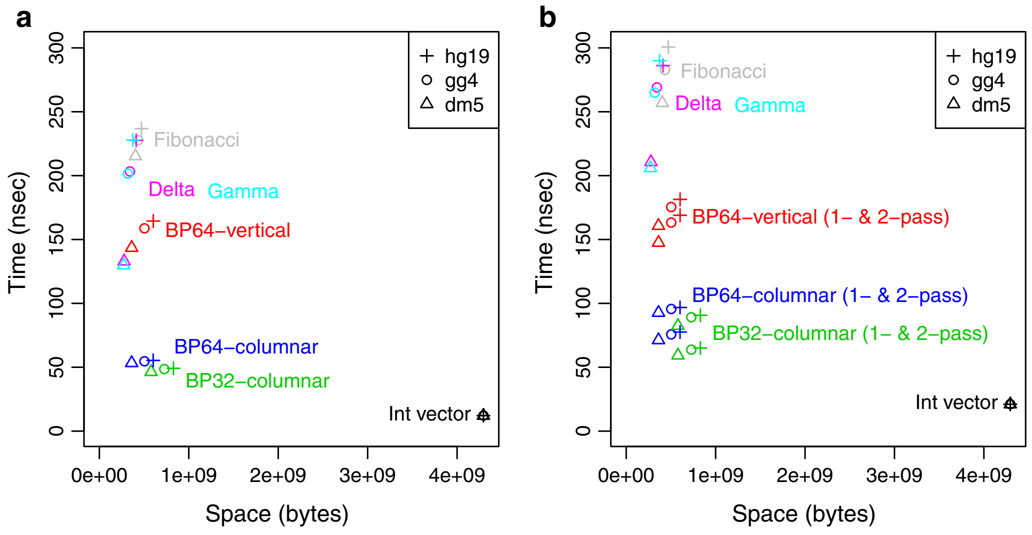


Fig. 6 Results for retrieval of offset array values. Space and time usage for retrieving offset array values under various compression schemes. Offsets are obtained from 15-mer hash tables for the fly (dm5), chicken (gg4), and human (hg19) genomes. *Graphs* show the retrieval time in nanoseconds per query as a function of the space required in bytes for each format. **a** Retrieval results for a single value. **b** Retrieval results for two adjacent values. Methods tested: storage as 32-bit integers (Int vector), or compressed using Elias gamma coding (Gamma), Elias delta (Delta), Fibonacci, BP64-vertical format, or the BP64-columnar and BP32-columnar schemes introduced here. For retrieving two adjacent values using vectorized bitpacking formats, results are shown for both two-pass (slower) and one-pass (faster) implementations

suggests that memory caching is more effective for the vertical format. This makes sense, because for the vertical format, the next adjacent value is likely to be decoded in the same group of four integers as the first value. But for the columnar format, an adjacent value requires another column of 128-bit registers to potentially be loaded and to be decoded.

For the vectorized bitpacking formats, we implemented one-pass methods, which retrieve two adjacent offset values by combining memory retrievals whenever possible. These methods can be contrasted with the two-pass approach that uses two separate calls to retrieve the desired values. As shown in Fig. 6b, the one-pass methods have a greater effect for the columnar formats, with the vertical format showing speedup by a factor of 1.07, but the columnar formats showing a speedup of 1.2–1.4. Overall, the one-pass methods for the BP64-columnar format are 2.1–2.2 times faster than the one-pass method for BP64-vertical, and the one-pass methods for the BP32-columnar format are 2.5–2.6 times faster. The comparison with the Elias gamma, Elias delta, and Fibonacci coding methods show a speedup of 2.9–3.9 times for BP64-columnar and 3.5–4.6 times for BP32-columnar, with the greater speedup seen for the chicken and human genomes. However, we should note that the SDSL methods were tested as two-pass retrievals, since no alternative one-pass implementations are available in the package.

Discussion

Our research in this area has been motivated largely by attempts to improve the space and time efficiency of our genomic alignment programs GMAP [7] and GSNAP [19]. Initially, both programs used a hash table of 12-mers. We subsequently increased the *k*-mer size to 15-mers, by compressing the hash table using an Elias gamma compression scheme (September 2011 release), which improved the speed of GSNAP by a factor of approximately 6–8. We then changed the bitpacking scheme to the BP64-vertical format and added a suffix array algorithm (October 2013 release), which gave an additional speedup of 4–7 times, depending on the similarity between the reads and the genome. The research in this paper represents our subsequent studies to improve the decoding speed of bitpacked data for random access, which we have used to further increase the speed of our programs with a BP64-columnar format (April 2014 release).

Our study leads to some insights for practical genomic implementations. For differential coding of offset arrays in hash tables, our BP64-columnar and BP32-columnar formats offer significant improvements in random-access decoding speed relative to the fastest known methods to date, with a threefold increase in speed for retrieving a single value and a twofold increase for two adjacent values. Between the two columnar alternatives, BP32-columnar gives a 12 % increase in speed for retrieving one value and a 20 % increase in speed for retrieving two adjacent values, but requires 40–60 % more space. Therefore, BP64-columnar appears to give a better balance between storage space and decoding speed, which underlies our decision to use this representation for our alignment programs.

Apart from this study, compression of small integers has not been widely applied or studied for genomic alignment. Some researchers [20] have used integer codes to compress the short reads themselves, after they have been aligned to a genome. Another application of integer codes has been to compress genomes or other nucleotide sequence datasets, either by themselves [21] or as differences relative to another genome [22].

In particular, although much research effort has gone into developing effective techniques for compressing genomic data structures [23, 24], little work has been done on compression techniques for hash tables, despite their prevalence in many bioinformatics applications. Hash tables are closely related to inverted index lists, which are used widely for indexing text and Web pages for rapid search [25], and where compression techniques have been applied [26]. However, those systems do not exploit the fact that a genomic hash table indexes an exhaustive set of *k*-mers, which presents domain-specific opportunities for compression.

An early system called CAFE [27] for aligning sequences to databases did use compression techniques in representing a genomic index. However, that system used an inverted index structure rather than a hash table, so it lacked an exhaustive set of offsets. Instead, it used integer coding to compress differences in the position table, achieving compression of 4–6 times smaller than the uncompressed size. We believe that compression of offsets is much more efficient than compressing positions, since for large genomes adjacent positions do not have the same pattern of closeness that offsets do and therefore do not compress as well. Moreover, to achieve maximal speed in combining hash table lookups, in procedures like the spanning set or complete set algorithms in GSNAP [19], it is faster to have lists of positions available in an uncompressed format.

One contribution of this paper has been to show how SIMD operations can be applied to genomic representations. Vectorization has found applications in other areas of bioinformatics, such as speeding up Smith-Waterman dynamic programming for nucleotide alignment [28–30] and profile HMM searches [31]. Likewise, we have applied vectorization to other parts of our alignment programs, such as computing mismatches between a query sequence and a genomic segment; constructing a localized hash table for a genomic region; performing dynamic programming alignment at the nucleotide level; and filtering results from a list of genomic positions to satisfy a given range of coordinates [32].

The BP64-columnar scheme in this paper is a solution to the general problem of differential coding with random access, and complements the BP128-vertical scheme for serial access. Therefore, our scheme conld be considered for any problem where a monotonically nondecreasing set of integers can be compressed and then retrieved by random access. In particular, many bitpacking schemes have been proposed for text retrieval [33], and database tables often have a format similar to our offset array. Applying our methodology to other domains is beyond the scope of this paper. However, by providing our methods in a standard package, we hope to facilitate further work by researchers to experiment with our methods and to use them more widely.

Additional files

Additional file 1. Source code. Source code in an archive format, using tar and bzip2, for all bitpacking implementations of hash tables, based on a modification of the SDSL 2.0 package. Package also includes the benchmarking programs used for timing experiments. This is the same package as provided in our companion paper on bitpacking for enhanced suffix arrays.**Additional file 2.** Source code for constructing benchmarks. Source code in an archive format, using tar and bzip2, for users to generate their own benchmarks for any genomic text in FASTA format. This is the same package as provided in our companion paper on bitpacking for enhanced suffix arrays.

Acknowledgements
The author thanks Daniel Lemire and Nathan Kurz for helpful discussions, and for making source code for their BP128 compression format publicly available. The author also thanks Simon Gog for modifying the SDSL functions to encode differences of zero efficiently.

Competing interests
The author declares that he has no competing interests.

Received: 3 November 2015 Accepted: 1 April 2016
Published online: 18 April 2016

References
1. Kahn SD. On the future of genomic data. Science. 2011;331:728–9.
2. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ. Basic local alignment search tool. J Mol Biol. 1990;215:403–10.
3. Ma B, Tromp J, Li M. PatternHunter: faster and more sensitive homology search. Bioinformatics. 2002;18:440–5.
4. Rumble SM, Lacroute P, Dalca AV, Flume M, Sidow A, Brudno M. SHRiMP: accurate mapping of color-space reads. PLoS Comput Biol. 2009;5:1000386.
5. Kent WJ. BLAT-the BLAST-like alignment tool. Genome Res. 2002;12:656–64.
6. Sedlazeck FJ, Rescheneder P, von Haeseler A. NextGenMap: fast and accurate read mapping in highly polymorphic genomes. Bioinformatics. 2013;29:2790–1.

7. Wu TD, Watanabe CK. GMAP: a genomic mapping and alignment program for mRNA and EST sequences. Bioinformatics. 2005;21:1859–975.
8. Ning Z, Cox AJ, Mullikin JC. SSAHA: a fast search method for large DNA databases. Genome Res. 2001;11:1725–9.
9. Ge H, Liu K, Juan T, Fang F, Newman M, Hoeck W. FusionMap: detecting fusion genes from next-generation sequencing data at base-pair resolution. Bioinformatics. 2011;27:1922–8.
10. Li M, Ma B, Kisman D, Tromp J. PatternHunter II: highly sensitive and fast homology search. Genome Inform. 2003;14:164–75.
11. Lemire D, Boytsov L. Decoding billions of integers per second through vectorization. Softw Pract Exp. 2015;45:1–29.
12. Elias P. Universal codeword sets and representations of the integers. IEEE Trans Inf Theory. 1975;21:194–203.
13. Fraenkel AS, Klein ST. Robust universal complete codes for transmission and compression. Discret Appl Math. 1996;64:31–55.
14. Willhalm T, Popovici N, Boshmaf Y, Plattner H, Zeier A, Schaffner J. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. Proc VLDB Endow. 2009;2:385–94.
15. Schlegel B, Gemulla R, Lehner W. Fast integer compression using SIMD instructions. In: Proceedings of the sixth international worshop on data management on new hardware. 2010. p. 34–40.
16. Lemire D, Boytsov L, Kurz N. SIMD compression and the intersection of sorted integers. Softw Pract Exp. 2015. doi:10.1002/spe.2326.
17. Shahbahrami A, Juurlink B, Vassiliadis S. Performance impact of misaligned accesses in SIMD extensions. In: Proceedings of 17th annual workshop on circuits, systems and signal processing. 2006. p. 334–42.
18. Gog S, Beller T, Moffat A, Petri M. From theory to practice: plug and play with succinct data structures. In: 13th international symposium on experimental algorithms. 2014. p. 326–37.
19. Wu TD, Nacu S. Fast and SNP-tolerant detection of complex variants and splicing in short reads. Bioinformatics. 2010;26:873–81.
20. Daily K, Rigor P, Christley S, Xie X, Baldi P. Data structures and compression algorithms for high-throughput sequencing technologies. BMC Bioinform. 2010;11:514.
21. Williams H, Zobel J. Compression of nucleotide databases for fast searching. CABIOS. 1997;13:549–54.
22. Brandon MC, Wallace DC, Baldi P. Data structures and compression algorithms for genomic sequence data. Bioinformatics. 2009;25:1731–8.
23. Giancarlo R, Scaturro D, Utro F. Textual data compression in computational biology: a synopsis. Bioinformatics. 2009;25:1575–86.
24. Deorowicz S, Grabowski S. Data compression for sequencing data. Algorithms Mol Biol. 2013;8:25.
25. Zobel J, Moffat A. Inverted files for text search engines. ACM Comput Surv. 2006;38:6.
26. Scholer F, Williams HE, Yiannis J, Zobel J. Compression of inverted indexes for fast query evaluation. In: Proceedings of the 25th annual ACM conference on research and development in information retrieval. 2002. p. 222–29.
27. Williams HE, Zobel J. Indexing and retrieval for genomic databases. IEEE Trans Knowl Data Eng. 2002;14:63–78.
28. Wozniak A. Using video-oriented instructions to speed up sequence comparison. Comput Appl Biosci. 1997;13:145–50.
29. Rognes T, Seeberg E. Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. Bioinformatics. 2000;16:699–706.
30. Farrar M. Striped Smith-Waterman speeds database searches six times over other simd implementations. Bioinformatics. 2007;23:156–61.
31. Eddy SR. Accelerated profile HMM searches. PLOS Comput Biol. 2011;7:1002195.
32. Wu TD, Reeder J, Lawrence M, Becker G, Brauer MJ. GMAP and GSNAP for genomic sequence alignment: Enhancements to speed, accuracy and functionality. Methods Mol Biol. 2016;1418:283–334.
33. Williams HE, Zobel J. Compressing integers for fast file access. Comput J. 1999;42:193–201.