Faster Population Counts Using AVX2 Instructions

WOJCIECH MUŁA, NATHAN KURZ AND DANIEL LEMIRE*

*Université du Québec (TELUQ), Canada
Email: lemire@gmail.com

Counting the number of ones in a binary stream is a common operation in database, information-retrieval, cryptographic and machine-learning applications. Most processors have dedicated instructions to count the number of ones in a word (e.g., `popcnt` on x64 processors). Maybe surprisingly, we show that a vectorized approach using SIMD instructions can be twice as fast as using the dedicated instructions on recent Intel processors. The benefits can be even greater for applications such as similarity measures (e.g., the Jaccard index) that require additional Boolean operations. Our approach has been adopted by LLVM: it is used by its popular C compiler (Clang).

Keywords: Software Performance; SIMD Instructions; Vectorization; Bitset; Jaccard Index

1. PINTRODUCTION

We can represent all sets of integers in $\{0, 1, \dots, 63\}$ using a single 64-bit word. For example, the word `0xAA` (`0b10101010`) represents the set $\{1, 3, 5, 7\}$. Intersections and unions between such sets can be computed using a single bitwise logical operation on each pair of words (AND, OR). We can generalize this idea to sets of integers in $\{0, 1, \dots, n-1\}$ using $\lceil n/64 \rceil$ 64-bit words. We call such data structures *bitsets*; they are also known as a bit vectors, bit arrays or bitmaps. Bitsets are ubiquitous in software, found in databases [1], version control systems [2], search engines [3, 4, 5], and so forth. Languages such as Java and C++ come with their own bitset classes (`java.util.BitSet` and `std::bitset` respectively).

The cardinality of a bitset (the number of one bits, each representing an element in the set) is commonly called a population count, a popcount, a Hamming weight, a sideways addition, or sideways sum. For example, the population counts of the words `0xFFFF`, `0xAA` and `0x00` are 16, 4 and 0 respectively. A frequent purpose for the population count is to determine the size of the intersection or union between two bitsets. In such cases, we must first apply a logical operation on pairs of words (AND, OR) and then compute the population count of the resulting words. For example, the cardinality of the intersection of the sets $A = \{4, 5, 6, 7\}$ and $B = \{1, 3, 5, 7\}$ represented by the words `0xF0` and `0xAA` can be computed as $|A \cap B| = \text{popcount}(\texttt{0xF0}\ \text{AND}\ \texttt{0xAA}) = \text{popcount}(\texttt{0xA0}) = 2$.

Population-count functions are used in cryptography [6], e.g., as part of randomness tests [7] or to generate pseudo-random permutations [8]. They can help find duplicated web pages [9]. They are frequently used in bioinformatics [10, 11, 12], ecology [13], chemistry [14], and so forth. Gueron and Krasnov use population-count instructions as part of a fast sorting algorithm [15].

The computation of the population count is so important that commodity processors have dedicated instructions: `popcnt` for x64 processors and `cnt` for the 64-bit ARM architecture.[1] The x64 `popcnt` instruction is fast: on recent Intel processors, it has a throughput of one instruction per cycle [17] (it can execute once per cycle) and a latency of 3 cycles (meaning that the result is available for use on the third cycle after execution). It is available in common C and C++ compilers as the intrinsic `_mm_popcnt_u64`. In Java, it is available as the `Long.bitCount` intrinsic.

Commodity PC processors also support Single-Instruction-Multiple-Data (SIMD) instructions. Starting with the Haswell microarchitecture (2013), Intel processors support the AVX2 instruction set which offers rich support for 256-bit vector registers. The contest between a dedicated instruction operating on 64-bits at a time (`popcnt`) and a series of vector instructions operating on 256-bits at a time (AVX2) turns out to be interesting. In fact, we show that we can achieve twice the speed of the an optimized `popcnt`-based function using AVX2: 0.52 versus 1.02 cycles per 8 bytes on large

[1]The x64 `popcnt` instruction was first available in the Nehalem microarchitecture, announced in 2007 and released in November 2008. The ARM `cnt` instruction was released as part of the Cortex-A8 microarchitecture, published in 2006 [16].

arrays. Our claim has been thoroughly validated: at least one major C compiler (LLVM's Clang) uses our technique [18].

Thus, in several instances, SIMD instructions might be preferable to dedicated non-SIMD instructions if we are just interested in the population count of a bitset. But what if we seek the cardinality of the intersection or union, or simply the Jaccard index between two bitsets? Again, the AVX2 instructions prove useful, more than doubling the speed (2.4×) of the computation against an optimized function using the `popcnt` instruction.

2. EXISTING ALGORITHMS AND RELATED WORK

Conceptually, one could compute the population count by checking the value of each bit individually by calling `count += (word >> i) & 1` for `i` ranging from 0 to 63, given that `word` is a 64-bit word. While this approach scales linearly in the number of input words, we expect it to be slow since it requires multiple operations for each bit in each word. It is $O(n)$—the best that we can do—but with a high constant factor.

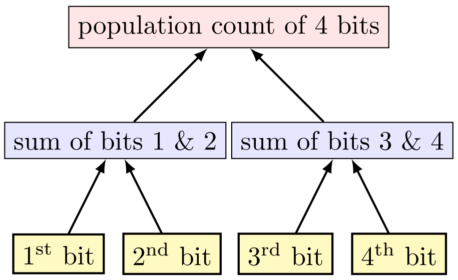

FIGURE 1. A tree of adders to compute the population count of four bits in two steps.

Instead, we should prefer approaches with fewer operations per word. We can achieve the desired result with a tree of adders and bit-level parallelism. In Fig. 1, we illustrate the idea over words of 4 bits (for simplicity). We implement this approach with two lines of code.

1. We can sum the individual bits to 2-bit subwords with the line of C code: `( x & 0b0101 ) + ( ( x >> 1 ) & 0b0101 )`. This takes us from the bottom of the tree to the second level. We say to this step exhibits *bit-level parallelism* since two sums are executed at once, within the same 4-bit word.
2. We can then sum the values stored in the 2-bit subwords into a single 4-bit subword with another line of C code: `( x & 0b0011 ) + ( ( x >> 2 ) & 0b0011 )`.

Fig. 2 illustrates a non-optimized (naive) function that computes the population count of a 64-bit word in this manner.

A fast and widely used tree-of-adder function to compute the population count has been attributed by

```
uint64_t c1  = UINT64_C(0x5555555555555555);
uint64_t c2  = UINT64_C(0x3333333333333333);
uint64_t c4  = UINT64_C(0x0F0F0F0F0F0F0F0F);
uint64_t c8  = UINT64_C(0x00FF00FF00FF00FF);
uint64_t c16 = UINT64_C(0x0000FFFF0000FFFF);
uint64_t c32 = UINT64_C(0x00000000FFFFFFFF);

uint64_t count(uint64_t x) {
  x = (x & c1) + ((x >> 1) & c1);
  x = (x & c2) + ((x >> 2) & c2);
  x = (x & c4) + ((x >> 4) & c4);
  x = (x & c8) + ((x >>  8)  & c8);
  x = (x & c16)+ ((x >> 16)) & c16);
  return (x & c32) + ((x >> 32) & c32);
}
```

FIGURE 2. A naive tree-of-adders function in C

```
uint64_t c1  = UINT64_C(0x5555555555555555);
uint64_t c2  = UINT64_C(0x3333333333333333);
uint64_t c4  = UINT64_C(0x0F0F0F0F0F0F0F0F);

uint64_t count(uint64_t x) {
  x -= (x >> 1) & c1;
  x = (( x >> 2) & c2) + (x & c2);
  x = ( x  + (x >> 4) ) & c4;
  x *= UINT64_C(0x0101010101010101);
  return x >> 56;
}
```

FIGURE 3. The Wilkes-Wheeler-Gill function in C

Knuth [19] to a 1957 textbook by Wilkes, Wheeler and Gill [20]: see Fig. 3. It involves far fewer than 64 instructions and we expect it to be several times faster than a naive function checking the values of each bit and faster than the naive tree-of-adder approach on processor with a sufficiently fast 64-bit integer multiplication (which includes all x64 processors).

- The first two lines in the `count` function correspond to the first two levels of our simplified tree-of-adders `count` function from Fig. 1. The first line has been optimized. We can verify the optimization by checking that for each possible 2-bit word, we get the sum of the bit values:
  - `0b11 - 0b01 = 0b10 = 2`,
  - `0b10 - 0b01 = 0b01 = 1`,
  - `0b01 - 0b00 = 0b01 = 1`,
  - `0b00 - 0b00 = 0b00 = 0`.
- After the first two lines, we have 4-bit population counts (in $\{0b0000, 0b0001, 0b0010, 0b0011, 0b0100\}$) stored in 4-bit subwords. The next line sums consecutive 4-bit subwords to bytes. We use the fact that the most significant bit of each 4-bit subword is zero.
- The multiplication and final shift sum all bytes in an efficient way. Multiplying `x` by `0x0101010101010101` is equivalent to summing up `x`, `x << 8`, `x << 16`, . . . , `x << 56`. The total population count is less than 64, so that the sum of all bytes from `x` fits in a single byte value (in $[0, 256)$). In that case, the most significant 8 bits from the product is the sum of all eight byte values.

Knuth also attributes another common technique to Wegner [21] (see Fig. 4) that could be competitive when the population count is relatively low (e.g., less than 4 one bit per 64-bit word). When the population count is expected to be high (e.g., more than 60 one bit per 64-bit words), one could simply negate the words prior to using the function so as to count the number of zeros instead. The core insight behind the Wegner function is that the line of C code `x &= x - 1` sets to zero the least significant bit of `x`, as one can readily check. On an x64 processor, the expression `x &= x - 1` might be compiled to the `blsr` (reset lowest set bit) instruction. On current generation processors, this instruction achieves a throughput of two instructions per cycle with a latency of one cycle [17]. The downside of the Wegner approach for modern processors is that the unpredictable loop termination adds a mispredicted branch penalty of at least 10 cycles [22], which for short loops can be more expensive than the operations performed by the loop.

```
int count(uint64_t x) {
  int v = 0;
  while(x != 0) {
    x &= x - 1;
    v++;
  }
  return v;
}
```

FIGURE 4. The Wegner function in C.

Another simple and common technique is based on tabulation. For example, one might create a table that contains the corresponding population count for each possible byte value, and then look up and sum the count for each byte. Such a table would require only 256 bytes. A population count for a 64-bit word would require only eight table look-ups and seven additions. On more powerful processor, with more cache, it might be beneficial to create a larger table, such as one that has a population count for each possible `short` value (2 bytes) using 64 KB. Each doubling of the bit-width covered by the table halves the number of table lookups, but squares the memory required for the table.

We can improve the efficiency of tree-of-adders techniques by *merging* the trees across words [23]. To gain an intuition for this approach, consider that in the Wilkes-Wheeler-Gill approach, we use 4-bit subwords to store the population count of four consecutive bits. Such a population count takes a value in $\{0, 1, 2, 3, 4\}$, yet a 4-bit integer can represent all integers in $[0, 16)$. Thus, as a simple optimization, we could accumulate the 4-bit counts across three different words instead of a single one. Next consider that if you sum two 4-bit subwords (representing integers in $[0, 16)$) the result is in $[0, 32)$ whereas an 8-bit subword (a byte) can represent all integers in $[0, 256)$, a range that is four times larger. Hence, we can accumulate the counts over four triple of words. These two optimizations combined lead to a function to compute the population count of twelve words at once (see Fig. 5) faster than would be possible if we processed each word individually.

```
uint64_t count(uint64_t *input) {
 uint64_t m1 = UINT64_C(0x5555555555555555);
 uint64_t m2 = UINT64_C(0x3333333333333333);
 uint64_t m4 = UINT64_C(0x0F0F0F0F0F0F0F0F);
 uint64_t m8 = UINT64_C(0x00FF00FF00FF00FF);
 uint64_t m16= UINT64_C(0x0000FFFF0000FFFF);
 uint64_t acc = 0;
 for (int j = 0; j < 12; j += 3) {
    uint64_t count1  =  input[j + 0];
    uint64_t count2  =  input[j + 1];
    uint64_t half1   =  input[j + 2];
    uint64_t half2   =  input[j + 2];
    half1  &=  m1;
    half2   = (half2  >> 1) & m1;
    count1 -= (count1 >> 1) & m1;
    count2 -= (count2 >> 1) & m1;
    count1 +=  half1;
    count2 +=  half2;
    count1  = (count1 & m2)
              + ((count1 >> 2) & m2);
    count1 += (count2 & m2)
              + ((count2 >> 2) & m2);
    acc  += (count1 & m4)
              + ((count1 >> 4) & m4);
 }
 acc = (acc & m8) + ((acc >>  8)  & m8);
 acc = (acc       +  (acc >> 16)) & m16;
 acc =  acc       +  (acc >> 32);
 return acc;
}
```

FIGURE 5. The Lauradoux population count in C for sets of 12 words.

However, even before Lauradoux proposed this improved function, Warren [24] had presented a superior alternative attributed to a newsgroup posting from 1997 by Seal, inspired from earlier work by Harley. This approach, henceforth called Harley-Seal, is based on a carry-save adder (CSA). Suppose you are given three bit values ($a, b, c \in \{0, 1\}$) and you want to compute their sum ($a + b + c \in \{0, 1, 2, 3\}$). Such a sum fits in a 2-bit word. The value of the least significant bit is given by $(a \operatorname{XOR} b) \operatorname{XOR} c$ whereas the most significant bit is given by $(a \operatorname{AND} b) \operatorname{OR}((a \operatorname{XOR} b) \operatorname{AND} c)$. Table 1 illustrates these expressions: the least significant bit $((a \operatorname{XOR} b) \operatorname{XOR} c)$ takes value 1 only when $a + b + c$ is odd and the most significant bit takes value 1 only when two or three of the input bits (a, b, c) are set to 1. There are many possible expressions to compute the most significant bit, but the chosen expression is convenient because it reuses the $a \operatorname{XOR} b$ expression from the computation of the least significant bit. Thus, we can sum three bit values to a 2-bit counter using 5 logical operations. We can generalize this approach to work on all 64-bits in parallel. Starting with three 64-bit input words, we can generate two new output words: h, which holds the 64 most significant bits, and l, which contains the corresponding 64 least significant bits. We effectively compute 64 sums in parallel using bit-level parallelism. Fig. 6 presents an efficient implementation in C of this idea. The function uses 5 bitwise logical operations (two XORs, two ANDs and one OR): it is optimal with respect to the number of such operations [25, 7.1.2]. However, it requires at least three cycles to complete due to data dependencies.

```
void CSA(uint64_t* h, uint64_t* l,
    uint64_t a, uint64_t b, uint64_t c) {
  uint64_t u = a ^ b;
  *h = (a & b) | (u & c);
  *l = u ^ c;
}
```

FIGURE 6. A C function implementing a bitwise parallel carry-save adder (CSA). Given three input words a, b, c, it generates two new words h, l in which each bit represents the high and low bits in the bitwise sum of the bits from a, b, and c.

From such a CSA function, we can derive an efficient population count. Suppose we start with three words serving as counters (initialized at zero): one for the least significant bits (henceforth `ones`), another one for the second least significant bits (`twos`, so named because each bit set represents 2 input bits), and another for the

TABLE 1. Sum of three bits $a + b + c$. We use $\oplus$ for XOR, $\wedge$ for AND and $\vee$ for OR.

a	b	c	$a+b+c$	$(a \oplus b) \oplus c$	$(a \wedge b) \vee ((a \oplus b) \wedge c)$
0	0	0	0	0	0
0	0	1	1	1	0
0	1	0	1	1	0
1	0	0	1	1	0
0	1	1	2	0	1
1	0	1	2	0	1
1	1	0	2	0	1
1	1	1	3	1	1

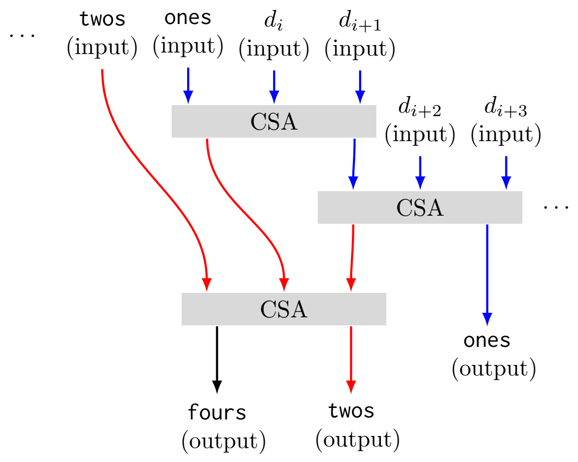

FIGURE 7. Harley-Seal algorithm aggregating four new inputs (d_i, d_{i+1}, d_{i+2}, d_{i+3}) to inputs `ones` and `twos`, producing new values of `ones`, `twos` and `fours`.

third least significant bits (`fours`, representing 4 input bits). We can proceed as follows; the first few steps are illustrated in Fig. 7. We start with a word serving as a population counter c (initialized at zero). Assume with we have a number of words $d_1, d_2, \ldots$ divisible by 8. Start with $i = 0$.

- Load two new words (d_i, d_{i+1}). Use the CSA function to sum `ones`, d_i and d_{i+1}, write the least significant bit of the sum to `ones` and store the carry bits in a temporary register (noted `twosA`). We repeat with the next two input words. Load d_{i+2}, d_{i+3}, use the CSA function to sum `ones`, d_i and d_{a+i}, write the least significant bit of the sum to `ones` and store the carry bits in a temporary register (noted `twosB`).
- At this point, we have three words containing second least significant bits (`twos`, `twosA`, `twosB`). We sum them up using a CSA, writing back the result to `twos` and the carry bits to a temporary register `foursA`.
- We do with d_{i+4}, d_{i+5} and d_{i+6}, d_{i+7} as we did with d_i, d_{i+1} and d_{i+2}, d_{i+3}. Again we have three words containing second least significant bits (`twos`, `twosA`, `twosB`). We sum them up with CSA, writing the result to `twos` and to a carry-bit temporary register `foursB`.
- At this point, we have three words containing third least significant bits (`fours`, `foursA`, `foursB`). We can sum them up with a CSA, write the result back to `fours`, storing the carry bits in a temporary register `eights`.
- We compute the population count of the word `eights` (e.g, using the Wilkes-Wheeler-Gill population count) and increment the counter c by the population count.
- Increment i by 8 and continue for as long as we have new words.

When the algorithm terminates, multiply c by 8. Compute the population count of `fours`, multiply the result by 4 and add to c. Do similarly with `twos` and `ones`. The counter c contains the population count. If the number of input words is not divisible by 8, adjust accordingly with the leftover words (e.g, using the Wilkes-Wheeler-Gill population count).

In that particular implementation of this idea, we used blocks of eight words. More generally, the Harley-Seal approach works with blocks of 2^n words for $n = 3, 4, 5, \ldots$ (8, 16, 32, ...). We need $2^n - 1$ CSA function calls when using 2^n words, and one call to an auxiliary function (e.g., Wilkes-Wheeler-Gill). If we expect the auxiliary function to be significantly more expensive than the CSA function calls, then larger blocks should lead to higher performance, as long as we have enough input data and many available registers. In practice, we found that using blocks of sixteen words works well on current processors (see Fig. 8). This approach is only worthwhile if we have at least 16 input words (64-

bits/word $\times$ 16 words = 128 bytes).

```
uint64_t harley_seal(uint64_t * d,
      size_t size) {
  uint64_t total = 0, ones = 0, twos = 0,
    fours = 0, eights = 0, sixteens = 0;
  uint64_t twosA, twosB, foursA, foursB, eightsA,
     eightsB;
  for(size_t i = 0; i < size - size % 16;
    i += 16) {
   CSA(&twosA, &ones, ones, d[i+0], d[i+1]);
   CSA(&twosB, &ones, ones, d[i+2], d[i+3]);
   CSA(&foursA, &twos, twos, twosA, twosB);
   CSA(&twosA, &ones, ones, d[i+4], d[i+5]);
   CSA(&twosB, &ones, ones, d[i+6], d[i+7]);
   CSA(&foursB, &twos, twos, twosA, twosB);
   CSA(&eightsA, &fours, fours, foursA, foursB);
   CSA(&twosA, &ones, ones, d[i+8], d[i+9]);
   CSA(&twosB, &ones, ones, d[i+10],d[i+11]);
   CSA(&foursA, &twos, twos, twosA, twosB);
   CSA(&twosA, &ones, ones, d[i+12],d[i+13]);
   CSA(&twosB, &ones, ones, d[i+14],d[i+15]);
   CSA(&foursB, &twos, twos, twosA, twosB);
   CSA(&eightsB, &fours, fours, foursA,
      foursB);
   CSA(&sixteens, &eights, eights, eightsA,
      eightsB);
   total += count(sixteens);
  }
  total = 16 * total + 8 * count(eights)
      + 4 * count(fours) + 2 * count(twos)
      + count(ones);
  for(size_t i = size - size % 16 ; i < size; i++)
   total += count(d[i]);
  return total;
}
```

FIGURE 8. A C function implementing the Harley-Seal population count over an array of 64-bit words. The `count` function could be the Wilkes-Wheeler-Gill function.

The functions we presented thus far still have their uses when programming with high-level languages without convenient access to dedicated functions (e.g., JavaScript, Go) or on limited hardware. However, they are otherwise obsolete when a sufficiently fast instruction is available, as is the case on recent x64 processors with `popcnt`. The `popcnt` instruction has a reciprocal throughput[2] of one instruction per cycle. With a properly constructed loop, the load-`popcnt`-add sequence can be executed in a single cycle, allowing for a population count function that processes 64-bits per cycle.

2.1. Existing Vectorized Algorithms

To our knowledge, the first published vectorized population count on Intel processor was proposed by Muła in 2008 [26]. It is a vectorized form of tabulation on 4-bit subwords. Its key ingredient is the SSSE3 vector instruction `pshufb` (see Table 2). The `pshufb` instruction shuffles the input bytes into a new vector containing the same byte values in a (potentially) different order. It takes an input register v and a control mask m, treating both as vectors of sixteen bytes. Starting from $v_0, v_1, \ldots, v_{16}$, it outputs a new vector $(v_{m_0}, v_{m_1}, v_{m_2}, v_{m_3}, \ldots, v_{m_{15}})$ (assuming that $0 \leq m_i < 16$ for $i = 0, 1, \ldots, 15$). Thus, for example, if the mask m is $0, 1, 2, \ldots, 15$, then we have the identify function. If the mask m is $15, 14, \ldots, 0$, then the byte order is reversed. Bytes are allowed to be repeated in the output vector, thus the mask $0, 0, \ldots, 0$ would produce a vector containing only the first input byte, repeated sixteen times. It is a fast instruction with a reciprocal throughput and latency of one cycle on current Intel processors, yet it effectively "looks up" 16 values at once. In our case, we use a fixed input register made of the input bytes $0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4$ corresponding to the population counts of all possible 4-bit integers $0, 1, 2, 3, \ldots, 15$. Given an array of sixteen bytes, we can call `pshufb` once, after selecting the least significant 4 bits of each byte (using a bitwise AND) to gather sixteen population counts on sixteen 4-bit subwords. Next, we right shift by four bits each byte value, and call `pshufb` again to gather sixteen counts of the most significant 4 bits of each byte. We can sum the two results to obtain sixteen population counts, each corresponding to one of the sixteen initial byte values. See Fig. 9 for a C implementation. If we ignore loads and stores as well as control instructions, Muła's approach requires two `pshufb`, two `pand`, one `paddb`, and one `psrlw` instruction, so six inexpensive instructions to compute the population counts of sixteen bytes. The Muła algorithm requires fewer instructions than the part of Wilkes-Wheel-Gill that does the same work (see Fig. 3), but works on twice as many input bytes per iteration.

```
__m128i count_bytes(__m128i v) {
  __m128i lookup = _mm_setr_epi8
      (0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4);
  __m128i low_mask = _mm_set1_epi8(0x0f);
  __m128i lo = _mm_and_si128(v, low_mask);
  __m128i hi = _mm_and_si128(
       _mm_srli_epi16(v, 4), low_mask);
  __m128i cnt1 =
       _mm_shuffle_epi8(lookup, lo);
  __m128i cnt2 =
       _mm_shuffle_epi8(lookup, hi);
  return _mm_add_epi8(cnt1, cnt2);
}
```

FIGURE 9. A C function using SSE intrinsics implementing Muła's algorithm to compute sixteen population counts, corresponding to sixteen input bytes.

The `count_bytes` function from Fig. 9 separately computes the population count for each of the sixteen input bytes, storing each in a separate byte of the result. As each of these bytes will be in $[0, 8]$, we can sum the result vectors from up to 31 calls to `count_bytes` using the `_mm_add_epi8` intrinsic without risk of overflow before using the `psadbw` instruction (using the `_mm_sad_-epu8` intrinsic) to horizontally sum the individual bytes into two 64-bit counters. In our implementation, we

[2]The reciprocal throughput is the number of processor clocks it takes for an instruction to execute.

found it adequate to call the count_bytes function eight times between each call to psadbw.

Morancho observed that we can use both a vector approach, like Muła's, and the popcnt in a hybrid approach [27]. Morancho proposed a family of hybrid schemes that could be up to 22% faster than an implementation based on popcnt for sufficiently large input arrays.

3. NOVEL VECTORIZED ALGORITHMS

Starting with the Haswell microarchiture released in 2013, Intel processors support the AVX2 instruction set with 256-bit vectors, instead of the shorter 128-bit vectors. It supports instructions and intrinsics that are analogous to the SSE intrinsics (see Table 2).[3]

The Muła function provides an effective approach to compute population counts at a speed close to an x64 processor's popcnt instruction when using 128-bit vectors, but after upgrading to AVX2's 256-bit vectors, it becomes faster than functions using the popcnt instruction. We present the basis of such a function in Fig. 10 using AVX2 intrinsics; the AVX2 intrinsics are analogous to the SSE intrinsics (see Fig. 2). It returns a 256-bit word that can be interpreted as four 64-bit counts (each having value in $[0, 64]$). We can then add the result of repeated calls with the _mm256_add_-epi64 intrinsic to sum 64-bit counts.

```
__m256i count(__m256i v) {
  __m256i lookup =
   _mm256_setr_epi8(0, 1, 1, 2, 1, 2, 2, 3, 1, 2,
        2, 3, 2, 3, 3, 4, 0, 1, 1, 2, 1, 2, 2, 3,
        1, 2, 2, 3, 2, 3, 3, 4);
  __m256i low_mask = _mm256_set1_epi8(0x0f);
  __m256i lo =  = _mm256_and_si256(v, low_mask);
  __m256i hi = _mm256_and_si256(_mm256_srli_epi32
      (v, 4), low_mask);
  __m256i popcnt1 = _mm256_shuffle_epi8(lookup,
      lo);
  __m256i popcnt2 = _mm256_shuffle_epi8(lookup,
      hi);
  __m256i total = _mm256_add_epi8(popcnt1,popcnt2
      );
  return _mm256_sad_epu8(total,
      _mm256_setzero_si256());
}
```

FIGURE 10. A C function using AVX2 intrinsics implementing Muła's algorithm to compute the four population counts of the four 64-bit words in a 256-bit vector. The 32 B output vector should be interpreted as four separate 64-bit counts that need to be summed to obtain the final population count.

For a slight gain in performance, we can call the Muła function several times while skipping the call to _mm256_-sad_epu8, adding the byte values with _mm256_add_epi8 before calling _mm256_sad_epu8 once. Each time we call the Muła function, we process 32 input bytes and get 32 byte values in $[0, 8]$. We can add sixteen totals before calling _mm256_sad_epu8 to sum the results into four 64-bit words (since $8 \times 16 = 128 < 2^8$), thus processing a block of 512 bytes per call.[4]

Of all the non-vectorized (or *scalar*) functions, Harley-Seal approaches are fastest. Thus we were motivated to port the approach to AVX2. The carry-save adder that worked on 64-bit words (see Fig. 6) can be adapted in a straight-forward manner to work with AVX2 intrinsics (see Fig. 11).

```
void CSA(__m256i* h, __m256i* l, __m256i a,
     __m256i b, __m256i c) {
  __m256i u = _mm256_xor_si256(a , b);
  *h = _mm256_or_si256(_mm256_and_si256(a , b) ,
       _mm256_and_si256(u , c) );
  *l = _mm256_xor_si256(u , c);
}
```

FIGURE 11. A C function using AVX2 intrinsics implementing a bitwise parallel carry-save adder (CSA).

Fig. 12 presents an efficient Harley-Seal function using an AVX2 carry-save adder. The function processes the data in blocks of sixteen 256-bit vectors (512 B). It calls Muła's AVX2 function (see Fig. 10).

Processors execute complex machine instructions using low-level instructions called μops.

- Using the dedicated popcnt instruction for the population of an array of words requires loading the word (movq), counting the bits (popcnt), and then adding the result to the total (addq). The load and the popcnt can be combined into a single assembly instruction, but internally they are executed as separate μops, and thus each 64-bit word requires three μops. Apart from minimal loop overhead, these three operations can be executed in a single cycle on a modern x64 superscalar processor, for a throughput of just over one cycle per 8 B word.
- The AVX2 Harley-Seal function processes sixteen 256-bit vectors (512 B) with 98 μops: 16 loads (vpmov), 32 bitwise ANDs (vpand), 15 bitwise ORs (vpor), and 30 bitwise XORs (vpxor). Each 64-bit word (8 B) thus takes just over 1.5 μops—about half as many as required to use the builtin popcnt instruction on the same input.

While fewer μops does does not guarantee faster execution, for computationally intensive tasks such as this it often proves to be a significant advantage. In this case, we find that it does in fact result in approximately twice the speed.

4. BEYOND POPULATION COUNTS

In practice, we often want to compute population counts on the result of some operations. For example, given two bitsets, we might want to determine the cardinality

[3]To our knowledge, Muła was first to document the benefits of AVX2 for the population count problem in March 2016 [26].

[4]We could call the Muła function up to 31 times, since $8 \times 31 = 248 < 2^8$.

TABLE 2. Relevant SSE instructions with latencies and reciprocal throughput in CPU cycles on recent (Haswell) Intel processors .

instruction	C intrinsic	description	latency	rec. throughput
paddb	_mm_add_epi8	add sixteen pairs of 8-bit integers	1	0.5
pshufb	_mm_shuffle_epi8	*shuffle* sixteen bytes	1	1
psrlw	_mm_srli_epi16	shift right eight 16-bit integers	1	1
pand	_mm_and_si128	128-bit AND	1	0.33
psadbw	_mm_sad_epu8	sum of the absolute differences of the byte values to the low 16 bits of each 64-bit word	5	1

```
uint64_t avx_hs(__m256i* d, uint64_t size) {
 __m256i total   = _mm256_setzero_si256();
 __m256i ones   = _mm256_setzero_si256();
 __m256i twos   = _mm256_setzero_si256();
 __m256i fours   = _mm256_setzero_si256();
 __m256i eights  = _mm256_setzero_si256();
 __m256i sixteens = _mm256_setzero_si256();
 __m256i twosA, twosB, foursA, foursB,
   eightsA, eightsB;
 for(uint64_t i = 0; i < size; i += 16) {
  CSA(&twosA, &ones, ones, d[i], d[i+1]);
  CSA(&twosB, &ones, ones, d[i+2], d[i+3]);
  CSA(&foursA, &twos, twos, twosA, twosB);
  CSA(&twosA, &ones, ones, d[i+4], d[i+5]);
  CSA(&twosB, &ones, ones, d[i+6], d[i+7]);
  CSA(&foursB,& twos, twos, twosA, twosB);
  CSA(&eightsA,&fours, fours, foursA,foursB);
  CSA(&twosA, &ones, ones, d[i+8], d[i+9]);
  CSA(&twosB, &ones, ones, d[i+10],d[i+11]);
  CSA(&foursA, &twos, twos, twosA, twosB);
  CSA(&twosA, &ones, ones, d[i+12],d[i+13]);
  CSA(&twosB, &ones, ones, d[i+14],d[i+15]);
  CSA(&foursB, &twos, twos, twosA, twosB);
  CSA(&eightsB, &fours, fours, foursA, foursB);
  CSA(&sixteens, &eights, eights, eightsA,
      eightsB);
  total=_mm256_add_epi64(total, count(sixteens));
 }
 total = _mm256_slli_epi64(total, 4);
 total = _mm256_add_epi64(total,
   _mm256_slli_epi64(count(eights), 3));
 total = _mm256_add_epi64(total,
   _mm256_slli_epi64(count(fours), 2));
 total = _mm256_add_epi64(total,
   _mm256_slli_epi64(count(twos),  1));
 total =_mm256_add_epi64(total,count(ones));
 return _mm256_extract_epi64(total, 0)
   + _mm256_extract_epi64(total, 1)
   + _mm256_extract_epi64(total, 2)
   + _mm256_extract_epi64(total, 3);
}
```

FIGURE 12. A C function using AVX2 intrinsics implementing Harley-Seal's algorithm. It assumes, for simplicity, that the input size in 256-bit vectors is divisible by 16. See Fig. 10 for the count function.

of their intersection (computed as the bit-wise logical AND) or the cardinality of their union (computed as the bit-wise logical OR). In such instances, we need to load input bytes from the two bitsets, generate a temporary word, process it to determine its population count, and so forth. When computing the Jaccard index, given that we have no prior knowledge of the population counts, we need to compute both the intersection and the union, and then we need to compute the two corresponding population counts (see Fig. 13).

```
void popcnt_jaccard_index(uint64_t* A, uint64_t*
    B, size_t n) {
  double s = 0;
  double i = 0;
  for(size_t k = 0; k < n; k++) {
    s += _mm_popcnt_u64(A[k] | B[k]);
    i += _mm_popcnt_u64(A[k] & B[k]);
   }
  return i / s;
}
```

FIGURE 13. A C function using the _mm_popcnt_u64 intrinsic to compute the Jaccard index of a pair of 64-bit inputs.

Both loads and logical operations benefit greatly from vectorization, and hybrid scalar/vector approaches can be difficult because inserting and extracting elements into and from vectors adds overhead. With AVX2, in one operation, we can load a 256-bit register or compute the logical AND between two 256-bit registers. This is four times the performance of the corresponding 64-bit operations. Thus we can expect good results from fast population count functions based on AVX2 adapted for the computation of the Jaccard index, the cardinality of the intersection or union, or similar operations.

5. EXPERIMENTAL RESULTS

We implemented our software in C. We use a Linux server with an Intel i7-4770 processor running at 3.4 GHz. This Haswell processor has 32 kB of L1 cache and 256 kB of L2 cache per core with 8 MB of L3 cache. The machine has 32 GB of RAM (DDR3-1600 with double-channel). We disabled Turbo Boost and set the processor to run at its highest clock speed. Our software is freely available (`https://github.com/CountOnes/hamming_weight`) and was compiled using the GNU GCC 5.3 compiler with the "`-O3 -march=native`" flags.

For all experiments, we use randomized input bits. However, we find that the performance results are insensitive to the data values.

Table 3 presents our results in number of cycles per word, for single-threaded execution. To make sure our results are reliable, we repeat each test 500 times and check that the minimum and the average cycle counts are within 1% of each other. We report the minimum cycle count divided by the number of words in the input. All the scalar methods (WWG, Laradoux, and HS) are significantly slower than the native `popcnt`-based function. We omit tabulation-based approaches from Table 3 because they are not competitive: 16-bit tabulation uses over 5 cycles even for large arrays. We can see that for inputs larger than 4 kB, the AVX2-based Harley-Seal approach is twice as fast as our optimized `popcnt`-based function, while for small arrays (fewer 64 words) the `popcnt`-based function is fastest.

We present the results for Jaccard index computations in Table 4. Contrary to straight population counts, the Jaccard-index AVX2 Muła remains faster than the `popcnt`-based function even for small blocks (256 B). AVX2 HS provides the best speed, requiring only 1.15 cycles to calculate the Jaccard similarity between each pair of 64-bit inputs. This is more than twice as fast (2.4×) as the `popcnt`-based function. Since the population count is done for each input of the pair, the speed of the similarity is only slightly greater than the speed of calculating the two population counts individually. That is, using AVX2 for both the Boolean operation and both population counts gives us the Boolean operation almost for free.

6. CONCLUSION

On recent Intel processors, the fastest approach to compute the population count on moderately large arrays (e.g., 4 kB) relies on a vectorized version of the Harley-Seal function. It is twice as fast as functions based on the dedicated instruction (`popcnt`). For the computation of similarity functions between two bitsets, a vectorized approach based on the Harley-Seal function is more than twice as fast (2.4×) as an optimized approach based on the `popcnt` instruction.

Future work should consider other computer architectures. Building on our work [26], Sun and del Mundo tested various population-count functions on an Nvidia GPU and found that its popcount intrinsic gave the best results [28].

ACKNOWLEDGEMENTS

We are grateful to S. Utcke for pointing out a typographical error in one of the figures.

FUNDING

This work was supported by the Natural Sciences and Engineering Research Council of Canada [261437].

REFERENCES

[1] Lemire, D., Ssi-Yan-Kai, G., and Kaser, O. (2016) Consistently faster and smaller compressed bitmaps with roaring. *Softw. Pract. Exp.*, **46**, 1547–1569.

[2] Martí, V. (2015). Counting objects. GitHub Engineering Blog, `http://githubengineering.com/counting-objects/` [last checked February 2017].

[3] Chambi, S., Lemire, D., Kaser, O., and Godin, R. (2016) Better bitmap performance with roaring bitmaps. *Softw. Pract. Exp.*, **46**, 709–719. spe.2325.

[4] Lemire, D., Boytsov, L., and Kurz, N. (2016) SIMD compression and the intersection of sorted integers. *Softw. Pract. Exp.*, **46**, 723–749.

[5] Grand, A. (2015). Frame of Reference and Roaring Bitmaps. `https://www.elastic.co/blog/frame-of-reference-and-roaring-bitmaps` [last checked February 2017].

[6] Hilewitz, Y., Shi, Z. J., and Lee, R. B. (2004) Comparing fast implementations of bit permutation instructions. *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Eighth Asilomar Conference on*, Washington, DC, USA, Nov, pp. 1856–1863. IEEE Computer Society.

[7] Suciu, A., Cobarzan, P., and Marton, K. (2011) The never ending problem of counting bits efficiently. *2011 RoEduNet International Conference 10th Edition*, Washington, DC, USA, Jun, pp. 1–4. IEEE Computer Society.

[8] Stefanov, E. and Shi, E. (2012) FastPRP: Fast Pseudo-Random Permutations for Small Domains. Technical Report 338. IACR Cryptology ePrint Archive.

[9] Manku, G. S., Jain, A., and Das Sarma, A. (2007) Detecting near-duplicates for web crawling. *Proceedings of the 16th International Conference on World Wide Web*, New York, NY, USA, May WWW '07, pp. 141–150. ACM.

[10] Prokopenko, D., Hecker, J., Silverman, E. K., Pagano, M., Nöthen, M. M., Dina, C., Lange, C., and Fier, H. L. (2016) Utilizing the Jaccard index to reveal population stratification in sequencing data: a simulation study and an application to the 1000 Genomes Project. *Bioinformatics*, **32**, 1366–1372.

[11] Lacour, A., Schüller, V., Drichel, D., Herold, C., Jessen, F., Leber, M., Maier, W., Noethen, M. M., Ramirez, A., Vaitsiakhovich, T., and Becker, T. (2015) Novel genetic matching methods for handling population stratification in genome-wide association studies. *BMC Bioinformatics*, **16**, 84.

[12] Li, Y., Patel, J. M., and Terrell, A. (2012) WHAM: a high-throughput sequence alignment method. *ACM T. Database Syst.*, **37**, 28.

[13] Dambros, C. S., Cáceres, N. C., Magnus, L., and Gotelli, N. J. (2015) Effects of neutrality, geometric constraints, climate, and habitat quality on species richness and composition of Atlantic forest small-mammals. *Global Ecol. Biogeogr.*, **24**, 1084–1093.

[14] Zhang, B., Vogt, M., Maggiora, G. M., and Bajorath, J. (2015) Design of chemical space networks using a Tanimoto similarity variant based upon maximum common substructures. *J. Comput. Aided Mol.*, **29**, 937–950.

TABLE 3. Number of cycles per 64-bit input word required to compute the population of arrays of various sizes.

array size	WWG	Lauradoux	HS	popcnt	AVX2 Muła	AVX2 HS
256 B	6.00	4.50	3.25	**1.12**	1.38	—
512 B	5.56	2.88	2.88	1.06	**0.94**	—
1 kB	5.38	3.62	2.66	1.03	0.81	**0.69**
2 kB	5.30	3.45	2.55	1.01	0.73	**0.61**
4 kB	5.24	3.41	2.53	1.01	0.70	**0.54**
8 kB	5.24	3.36	2.42	1.01	0.69	**0.52**
16 kB	5.22	3.36	2.40	1.01	0.69	**0.52**
32 kB	5.23	3.34	2.40	1.01	0.69	**0.52**
64 kB	5.22	3.34	2.40	1.01	0.69	**0.52**

TABLE 4. Number of cycles per pair of 64-bit input words required to compute the Jaccard index of arrays of various sizes.

array size	popcnt	AVX2 Muła	AVX2 HS
256 B	3.00	**2.50**	—
512 B	2.88	**2.00**	—
1 kB	2.94	2.00	**1.53**
2 kB	2.83	1.84	**1.33**
4 kB	2.80	1.76	**1.22**
8 kB	2.78	1.75	**1.16**
16 kB	2.77	1.75	**1.15**
32 kB	2.76	1.75	**1.15**
64 kB	2.76	1.74	**1.15**

[15] Gueron, S. and Krasnov, V. (2016) Fast quicksort implementation using AVX instructions, . **59**, 83–90.

[16] Hill, S. (2006) Design of a reusable 1GHz, superscalar ARM processor. *2006 IEEE Hot Chips 18 Symposium (HCS)*, Washington, DC, USA, Aug, pp. 1–18. IEEE Computer Society.

[17] Fog, A. (2016) Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. Technical report. Copenhagen University College of Engineering, Copenhagen, Denmark. `http://www.agner.org/optimize/instruction_tables.pdf` [last checked December 2016].

[18] Carruth, C. (2015). Implement a faster vector population count based on the PSHUFB. `https://reviews.llvm.org/rL238636` [last checked December 2016].

[19] Knuth, D. E. (2009) *Bitwise Tricks & Techniques*, The Art of Computer Programming, **4**. Addison-Wesley, Boston, Massachusetts.

[20] Wilkes, M. V., Wheeler, D. J., and Gill, S. (1957) *The Preparation of Programs for an Electronic Digital Computer*, second edition. Addison-Wesley Publishing, Boston, USA.

[21] Wegner, P. (1960) A technique for counting ones in a binary computer. *Commun. ACM*, **3**, 322–.

[22] Rohou, E., Swamy, B. N., and Seznec, A. (2015) Branch prediction and the performance of interpreters: Don't trust folklore. *Proc. 13th IEEE/ACM International Symposium on Code Generation and Optimization*, Washington, DC, USA, February CGO '15, pp. 103–114. IEEE Computer Society.

[23] Lauradoux, C. (2009). Hamming weight. (It was posted on the author's website, but is apparently no longer available online.).

[24] Warren, H. S., Jr. (2007) The quest for an accelerated population count. In Wilson, G. and Oram, A. (eds.), *Beautiful Code: Leading Programmers Explain How They Think*, chapter 10, pp. 147–160. O'Reilly Media, Sebastopol, California.

[25] Knuth, D. E. (2011) *Combinatorial Algorithms, Part 1*, The Art of Computer Programming, **4A**. Addison-Wesley, Boston, Massachusetts.

[26] Muła, W. SSSE3: fast popcount. `http://0x80.pl/articles/sse-popcount.html` [first published in 2008, revised in March 2016, last checked February 2016].

[27] Morancho, E. (2014) A hybrid implementation of Hamming weight. *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Sankt Augustin, Germany, Feb, pp. 84–92. Euromicro.

[28] Sun, C. and del Mundo, C. C. (2016) Revisiting POPCOUNT operations in CPUs/GPUs. ACM Student Research Competition Posters at the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis.

[29] Intel Corporation (2016). Intel Architecture Instruction Set Extensions Programming Reference. `https://intel.ly/2hurf64` [last checked February 2017].

APPENDIX A. POPULATION COUNTS IN AVX-512

Though Intel processors through the current Kaby Lake generation do not yet support the AVX-512 instruction set, it is straight-forward to generalize our vectorized algorithms to 512-bit vectors. However, even beyond the increase in vector width, it should be possible to implement the carry-save adder more efficiently with AVX-512, which also adds the `vpternlogd` instruction. Available through C intrinsics as `_mm512_ternarylogic_epi64`, this instruction allows us to compute arbitrary three-input binary functions in a single operation. Utilizing this, we can replace the 5 logical instructions we needed for AVX2 with just two instructions. The `vpternlogd` instruction relies on an integer parameter i that serves as a look-up table. Given the input bits x, y, z, the value given that the $(x + 2y + 4z)^{\text{th}}$ bit of the parameter i is returned. For example, to compute XOR of the inputs, the i parameter needs to have a 1-bit at indexes 1, 2, 4 and 7 (i.e., $i = 2^1 + 2^2 + 2^4 + 2^7 = 150$ or `0x96` in hexadecimal). Similarly, to compute the most significant bit of the carry-save adder, the i parameter needs to have a 1-bit at indexes 3, 5, 6 and 7 (`0xe8` in hexadecimal). Fig. A.1 presents a C function implementing a carry-save adder (CSA) using AVX-512 intrinsics.

```
void CSA(__m512i* h, __m512i* l, __m512i a,
    __m512i b, __m512i c) {
  *l = _mm512_ternarylogic_epi32(c, b, a, 0x96);
  *h = _mm512_ternarylogic_epi32(c, b, a, 0xe8);
}
```

FIGURE A.1. A C function using AVX-512 intrinsics implementing a bitwise parallel carry-save adder (CSA).

Intel has also announced that future processors might support the AVX-512 `vpopcnt` instruction [29] which computes the population count of each word in a vector. To our knowledge, no available processor supports `vpopcnt` at this time.