

# Branchless and SIMD Filtering of Hard Special Leaves Using a Compact Factor Table

Kim Walisch

March 31, 2026

## Abstract

This paper presents practical improvements for computing hard special leaves in the combinatorial prime counting algorithms of Lagarias–Miller–Odlyzko, Deléglise–Rivat, and Gourdon. We analyze a compact factor table representation, originally introduced by Christian Bau in 2003, as a data structure for identifying admissible square-free values and compare it with other approaches. We then derive a branchless scalar filtering algorithm and SIMD implementations using AVX512 and ARM SVE instructions. The resulting method combines reduced memory usage, sequential memory access, and efficient vectorization. Benchmarks on a recent set of CPUs show speedups of up to 32% in `primecount`’s hard special leaves implementation.

## 1 Introduction

The combinatorial prime counting algorithms of Lagarias–Miller–Odlyzko [1], Deléglise–Rivat [2], and Gourdon [3] decompose the computation of  $\pi(x)$  into several summation formulas. Among these, the formula for the so-called hard special leaves is the most computationally expensive and the most difficult to implement. In practice, the hard special leaves are typically computed using a prime sieve such as the segmented Sieve of Eratosthenes.

In these algorithms, a special leaf is an individual contribution of the form

$$-\mu(m)\phi\left(\frac{x}{p \cdot m}, \pi(p) - 1\right),$$

where  $p$  is prime,  $m$  is square-free with  $p_{\min}(m) > p$ , and  $\phi(z, a)$  counts integers  $n \leq z$  that are coprime to the first  $a$  primes. A special leaf is called hard if this remaining  $\phi$ -term cannot be evaluated in  $O(1)$  time and must instead be computed explicitly.

Algorithmically, hard special leaves are typically treated in two cases:

1. **Leaves composed of a prime and a square-free number:**  $n = p \cdot m$
2. **Leaves composed of two primes:**  $n = p \cdot q$

Both cases contribute to the same overall summation, but they behave quite differently computationally. The second case, products of two primes, can be processed using relatively simple nested loops and does not pose major implementation difficulties. The first case, products of a prime and a square-free number, is considerably more difficult because it requires efficient generation and filtering of admissible square-free values  $m$  together with evaluation of the corresponding  $\phi$ -terms.

This algorithmic separation already appears in the literature. In the Lagarias–Miller–Odlyzko [1] algorithm, special handling is provided for leaves of the form  $\frac{x}{pm}$ , while leaves of the form  $\frac{x}{pq}$  are treated separately with simpler methods. Similar decompositions appear in the Deléglise–Rivat [2] and Oliveira e Silva [4] implementations.

This paper focuses exclusively on the first case, namely the computation of hard special leaves of the form  $p \cdot m$ , where  $m$  is square-free.

## 2 Conditions that define admissible square-free values of $m$

In the combinatorial prime counting algorithms of Lagarias–Miller–Odlyzko (LMO) [1] and Deléglise–Rivat [2], the hard special leaves of the form  $n = p \cdot m$  are determined by the following admissibility conditions on  $m$  for a fixed prime  $p$ :

### 1. Square-free condition

$$\mu(m) \neq 0$$

where  $\mu(m)$  denotes the Möbius function of  $m$ . This condition ensures that  $m$  is square-free and that each product of distinct primes is counted exactly once in the inclusion–exclusion framework underlying the combinatorial algorithm.

### 2. Least prime factor condition

$$p_{\min}(m) > p$$

Equivalently, all prime factors of  $m$  are strictly larger than the current prime  $p$ . This condition enforces a canonical ordering of prime factors and avoids duplicate representations of the same integer.

## 3 Previous Implementation Strategies in the Literature

### 3.1 Lagarias–Miller–Odlyzko

The Lagarias–Miller–Odlyzko (LMO) algorithm [1] uses a preprocessing approach that enumerates only admissible square-free values  $m$ , thereby avoiding runtime filtering.

LMO constructs three families of lookup tables:

- $A_k$ : square-free numbers  $m \leq y$  with smallest prime factor  $p_k$ ,
- $M_k$ : corresponding Möbius values  $\mu(m)$ ,
- $N_k$ : index tables used to locate the relevant subranges in  $A_k$ .

These tables are built for all primes  $p_k \leq \sqrt{y}$  and together require  $O(y \log \log x)$  memory. The admissible values are partitioned into disjoint subsets

$$\{m\} = \bigsqcup_k A_k,$$

and processed one subset at a time. For each  $k$ , the table  $N_k$  is used to determine the relevant index range, after which the corresponding segment of  $A_k$  is traversed sequentially. Thus, only admissible square-free values are visited and no filtering is required inside the inner loop.

### 3.2 Oliveira e Silva

Oliveira e Silva [4] proposes a dense lookup table approach that replaces the precomputed sparse structures used in LMO by a single compact representation.

A lookup table is constructed for all integers  $m \leq y$ :

$$T[m] = \mu(m)p_{\min}(m).$$

Thus, this table encodes both square-freeness and the least prime factor in a single signed value. The admissibility condition can then be tested using a single comparison  $|T[m]| > p$ . Instead of generating only admissible values, the algorithm scans the interval  $m \in ]m_{\min}, m_{\max}]$  and filters it using the lookup table:

```
for (int64_t m = m_max; m > m_min; m--)
{
    if (abs(T[m]) > prime)
        process(x / (prime * m));
}
```

### 3.3 Staple

Staple [5] proposes an alternative strategy for enumerating square-free values  $m$  that avoids the use of dense lookup tables such as  $p_{\min}(m)$  and  $\mu(m)$ . In order to eliminate the array  $p_{\min}(y)$ , the square-free values  $m \in [m_{\min}, m_{\max}]$  that are coprime to the first  $b + 1$  primes can be generated recursively as products

$$m = p_{b_1}p_{b_2} \cdots p_{b_k},$$

subject to  $b + 1 < b_1 < b_2 < \cdots < b_k$  with  $m < m_{\max}$ . This ensures  $p_{\min}(m) > p_{b+1}$ , and values with  $m < m_{\min}$  are discarded. Since each value of  $m$  is constructed as a product of distinct primes, it is square-free by construction and satisfies  $\mu(m) = (-1)^k$  when  $m$  has  $k$  prime factors.

The algorithm requires only a table of primes up to  $y$ , resulting in a memory usage of  $O(y / \log y)$ , and eliminates the  $O(y)$  lookup tables used in dense table approaches.

### 3.4 Precomputed square-free list with range restriction

A similar approach is used in recent experimental implementations, such as the Rust V8 program developed by D. Ruiu [8] using an AI assisted optimization workflow. In this approach, a list of all square-free numbers  $m \leq y$  is precomputed, optionally restricted to values whose prime factors are coprime to the first few small primes. Instead of scanning all integers or generating square-free values recursively, the algorithm iterates over this precomputed list.

For each current prime  $p$ , the admissible range  $]m_{\min}, m_{\max}]$  is determined from the bounds of the formula, and the corresponding subrange of square-free values is located using binary search. The iteration then proceeds sequentially over this subrange, with the remaining admissibility condition  $p_{\min}(m) > p$  checked inside the loop.

Thus, the enumeration of  $m$  is reduced to a scan over a precomputed ordered list, with range restriction implemented via binary search. The memory usage of this approach is  $O(y)$ , as it stores all relevant square-free values up to  $y$ .

### 3.5 Compact factor table preview

The compact factor table developed in this paper occupies a middle ground between Oliveira e Silva’s dense lookup table and the sparse admissible-value structures used by the other approaches. Like Oliveira e Silva’s method [4], it scans a contiguous in-memory representation and applies a runtime admissibility test. However, instead of storing one entry for every integer  $m \leq y$ , it stores entries only for integers that survive a small wheel factorization and encodes square-freeness, Möbius sign, and least prime factor in a single compact value. As a result, it preserves the sequential access pattern that is favorable for modern CPUs while achieving a substantially smaller practical memory footprint than a dense table.

The full structure of the compact factor table is described in Section 4. For the purpose of comparison, however, its most important high level properties can already be stated here: it uses  $O(y)$  memory, scans a wheel compressed candidate set in monotonic order, and performs filtering using a single runtime comparison.

### 3.6 Comparison of approaches

The approaches described above, together with the compact factor table previewed in Section 3.5, differ primarily in how they represent and enumerate admissible square-free values  $m$ , and in the resulting trade-offs between memory usage, iteration cost, and implementation complexity.

A first important distinction is between methods that **enumerate only values admissible for the current interval** and those that **scan a larger set and filter at runtime**. The LMO algorithm targets only admissible values  $m$  for the current range. Staple’s recursive method avoids scanning all integers and generates square-free values satisfying the prime factor constraints, but values outside the current interval  $]m_{\min}, m_{\max}]$  may still need to be discarded during the traversal. By contrast, Oliveira e Silva’s approach scans all integers in the interval and uses a lookup table to filter admissible values. The precomputed square-free list approach lies between these extremes: it avoids scanning all integers, but still performs a per-element admissibility test within a restricted candidate set. The compact factor table is closest in spirit to Oliveira e Silva’s method, but it reduces the scanned set by storing only wheel-compressed candidates instead of all integers.

The methods also differ significantly in their **memory requirements**. Oliveira e Silva’s dense table requires  $O(y)$  memory, while Staple’s recursive approach requires only  $O(y/\log y)$  memory for the prime table. The precomputed square-free list requires  $O(y)$  memory to store all candidate values. The compact factor table also uses  $O(y)$  memory asymptotically, but its practical footprint is much smaller than that of Oliveira e Silva’s dense table because it combines wheel compression with a reduced element size. The LMO algorithm uses multiple table families and has the largest practical memory footprint among the approaches considered here.

A further important distinction concerns the **practical overhead of enumeration**. Methods that avoid scanning all integers incur additional computational overhead in order to restrict the iteration to admissible values. In the LMO algorithm, the interval  $]m_{\min}, m_{\max}]$  must be recomputed repeatedly for each prime using integer division, which is relatively expensive. In Staple’s recursive

approach, the traversal generates square-free values starting from small products, so that many values  $m \leq m_{\min}$  are produced and subsequently discarded. In addition, both approaches rely on non-sequential access patterns and irregular control flow, which are less favorable for modern CPU architectures compared to simple linear scans. The compact factor table instead supports a simple sequential scan after the interval bounds have been converted to compressed indexes, and its runtime filter reduces to a single comparison.

Another important property is whether the algorithm iterates over the admissible values  $m$  in **monotonic order**, i.e. strictly increasing or decreasing. This property is required by more advanced counting techniques, such as the method described in [6], where the positions of consecutive leaves must increase in order to allow efficient batch counting. Among the approaches considered here, the Oliveira e Silva method, the precomputed square-free list approach, and the compact factor table all iterate over values in monotonic order. By contrast, the LMO algorithm and Staple’s recursive method do not guarantee monotonic ordering of the square-free values  $m$ .

These trade-offs can be summarized as follows:

Approach	Memory usage (asymptotic)	Memory usage (practical)	Iteration set	Filtering	Memory access pattern
LMO	$O(y \log \log x)$	very high	only admissible $m$	none	non-sequential
Oliveira e Silva	$O(y)$	moderate	all integers $m$	admissibility test	sequential
Staple	$O(y/\log y)$	very low	square-free $m$	interval check	non-sequential
Precomputed square-free list	$O(y)$	low	precomputed square-free $m$	admissibility test	sequential
Compact factor table	$O(y)$	very low	wheel-coprime candidate $m$	admissibility test	sequential

## 4 Detailed Structure of the Compact Factor Table

In this section, we give a detailed description of the compact lookup table representation previewed in Section 3.5. The data structure itself is not new: it was originally introduced by Christian Bau in a 2003 unpublished paper and implemented in his accompanying extended Meissel–Lehmer code. It has also been used in the `primecount` C/C++ library [7] since around 2015. However, its properties have not been studied in detail in the literature. In particular, there has been no published paper explaining its memory advantages or showing how it can be used to obtain a branchless and SIMD-friendly filtering algorithm for hard special leaves.

The key idea of the compact factor table, as originally devised by Christian Bau, is to store factorization-related information only for integers that are coprime to a small set of primes using wheel factorization. This avoids storing one entry per integer and instead restricts the representation to a reduced residue system. The table also uses a smaller integer type than the primes being processed. For composite numbers this is sufficient because the stored least prime factor always

satisfies  $p_{\min}(n) \leq \sqrt{n}$ . Only primes themselves can exceed the table type, and those are encoded by the maximum table value as a sentinel. As a result, the memory footprint of this representation is significantly smaller than that of dense lookup tables such as those used in previous approaches.

The main contribution of this work is not the data structure itself, nor the use of a single-comparison predicate, both of which already appear in Christian Bau’s implementation. Rather, the contribution of this paper is to show how this representation can be used to obtain a branchless filtering algorithm for admissible square-free values  $m$ , and how this filtering step can be efficiently vectorized using SIMD instructions. This leads to a more efficient implementation of the hard special leaves algorithm on modern CPU architectures.

In the following subsections, we describe the structure of the compact factor table in detail, explain its memory advantages, and show how it forms the basis for the branchless scalar and SIMD filtering algorithms presented in Sections 6 and 7.

#### 4.1 Factor table structure and encoding

We write  $F(n)$  for the value stored by the compact factor table for the represented integer  $n$ .

Let  $T$  denote the integer type used by the table, and let  $T_{\max}$  be its largest value. In practice, a 16-bit factor table suffices for represented values up to about 32 bits, and a 32-bit factor table suffices for represented values up to about 64 bits. More precisely, since  $T_{\max} - 1$  and  $T_{\max}$  are reserved sentinel values, the safe bound is  $n \leq (T_{\max} - 1)^2 - 1$ . The stored value  $F(n)$  combines square-freeness, the Möbius sign, and the least prime factor into a single machine word.

The encoding is as follows:

Case	Stored value $F(n)$	Interpretation
$n = 1$	$\text{MAX}(T) - 1$	Special value for the neutral element; it also encodes $\mu(1) = 1$ .
$n$ prime	$\text{MAX}(T)$	Sentinel meaning “prime”; the true least prime factor is $n$ itself and may not fit into type $T$ .
$\mu(n) = 0$	0	$n$ is not square-free.
$\mu(n) = 1$	$p_{\min}(n) - 1$	$n$ is square-free and has an even number of prime factors.
$\mu(n) = -1$	$p_{\min}(n)$	$n$ is square-free and has an odd number of prime factors.

For Gourdon’s  $D$  formula, the implementation in `primecount` uses one additional convention that is not shown in the table above:  $F(n) = 0$  also when  $n$  has a prime factor  $q > y$ . Since the table is built only for values  $n \leq z$ , this excludes exactly the cases in which  $n$  contains a prime factor satisfying  $y < q \leq z$ . In this way, Gourdon’s additional admissibility condition is absorbed into the same compact encoding and can still be handled by the same single-comparison filter.

This encoding is convenient for two reasons. First, it stores all information needed by the hard special leaves filter in a single compact value. Second, for square-free composite numbers the least significant bit of  $F(n)$  encodes the sign of  $\mu(n)$ :  $F(n)$  is even when  $\mu(n) = 1$  and odd

when  $\mu(n) = -1$ . This works because every stored least prime factor is an odd prime, so replacing  $p_{\min}(n)$  by  $p_{\min}(n) - 1$  changes only the lowest bit.

The special values  $\text{MAX}(T) - 1$  and  $\text{MAX}(T)$  are chosen so that the same representation also handles the exceptional cases  $n = 1$  and  $n$  prime without requiring a second table. The practical effect of this encoding can already be seen in the classical inner loop for hard special leaves of the form  $n = pm$ . In the source code examples below the  $p_{\min}(m)$  lookup table is named `lpf(m)`.

Using separate  $\mu(m)$  and `lpf(m)` lookup tables, the LMO/DR admissibility test requires two conditions (for Gourdon, an additional third condition is required):

```
for (int64_t m = m_max; m > m_min; m--)
{
    if (mu[m] != 0 && lpf[m] > prime)
        process(x / (prime * m));
}
```

With the compact factor table, the same test reduces to a single comparison:

```
// Simplified example, ignoring wheel compression
for (int64_t m = m_max; m > m_min; m--)
{
    if (factor_table[m] > prime)
        process(x / (prime * m));
}
```

In conceptual form, this is the key benefit of the encoding: a two-part predicate involving  $\mu(m)$  and `lpf(m)` is replaced by a single integer comparison. In an actual implementation, the Möbius sign can also be recovered from the same encoded table value, so no separate  $\mu$  table is required. This simplification of the hot predicate is what later enables the branchless filtering and SIMD vectorization described in Sections 6 and 7.

## 4.2 Memory reduction using wheel factorization

The primary source of space reduction is wheel factorization. In practice, a relatively small wheel  $W$  is chosen so that the auxiliary lookup tables used for converting integers to compact array indexes and back remain small enough to stay in the CPU's fast cache memory, ideally in the L1 cache. Christian Bau's original 2003 implementation of the factor table data structure supported both modulo 30 and modulo 210 wheels, selectable at compile time. The `primecount` C/C++ library [7] uses the larger but still cache friendly wheel

$$W = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 = 2310.$$

The factor table then stores entries only for integers  $n$  with  $\gcd(n, W) = 1$ . Thus, all numbers divisible by one of the first five primes are omitted from the array from the outset. Since  $\varphi(2310) = 480$ , only 480 residue classes remain in each block of 2310 consecutive integers. Hence the number of stored entries is reduced by the factor  $2310/480 = 4.8125$ . This is not only a space reduction but also a constant factor performance benefit: any linear scan over candidate values  $m$  touches only the represented residues, so the amount of filtering work is reduced by the same factor.

Conceptually, the factor table can be viewed as a sieve of Eratosthenes array with wheel factorization applied. A dense sieve would represent every integer, whereas the factor table used in `primecount` represents only integers that are not divisible by 2, 3, 5, 7, or 11. This point of view is useful because the table is initialized by a modified sieve of Eratosthenes. The special entry for  $n = 1$  is first set to  $\text{MAX}(T) - 1$ , and all other represented entries are initialized to  $\text{MAX}(T)$ . The sieve then iterates over primes  $p$  and visits only represented multiples of  $p$ . On the first visit,  $p$  is stored as the least prime factor. On later visits, the least significant bit is toggled in order to record the parity of the number of distinct prime factors. Finally, multiples of  $p^2$  are set to 0, thereby marking the non-square-free integers for which  $\mu(n) = 0$ . Entries already set to 0 are never toggled again.

Wheel compression requires a conversion between ordinary integers and compact array indexes. Let  $c_0, c_1, \dots, c_{479}$  denote the residue classes modulo 2310 that are coprime to 2310, listed in increasing order, and define

$$\rho(r) = \begin{cases} -1, & r = 0, \\ \max\{j : c_j \leq r\}, & 1 \leq r < 2310. \end{cases}$$

Then, for any positive integer  $n$ , write  $n = 2310q + r$  with  $0 \leq r < 2310$ . The mapping  $\text{index}(n) = 480q + \rho(r)$  returns the index of the largest represented integer  $\leq n$ . In particular, if  $\gcd(r, 2310) = 1$ , then  $n$  itself is represented. This is the behavior needed when interval bounds such as  $m_{\min}$  and  $m_{\max}$  are converted into compact indexes. Conversely, if  $i = 480q + j$ , then the represented integer stored at table index  $i$  is  $\text{number}(i) = 2310q + c_j$ .

The following code implements the mapping described above:

```
static int64_t to_index(uint64_t number)
{
    uint64_t q = number / 2310;
    uint64_t r = number % 2310;
    return 480 * q + coprime_indexes[r];
}

static int64_t to_number(uint64_t index)
{
    uint64_t q = index / 480;
    uint64_t r = index % 480;
    return 2310 * q + coprime[r];
}
```

In the implementation, these forward and inverse mappings are realized using two small static lookup tables. They are used whenever the algorithm moves between arithmetic formulas written in terms of the integer  $m$  and the compact in-memory representation used by the factor table. For example, interval bounds such as  $m_{\min}$  and  $m_{\max}$  are converted into compact indices before scanning the table, while the inverse mapping is used when an index must be converted back into the corresponding value of  $m$ , for example when evaluating  $x/(p \cdot m)$ .



## 5 SIMD Suitability of Different Data Structures

Not every data structure discussed in Section 3 is equally suitable for the branchless SIMD filtering algorithm developed later in this paper. That algorithm assumes a monotonic scan over a dense in-memory representation and applies the same admissibility test to many consecutive candidates in parallel.

Using the LMO lookup tables  $A_k$ ,  $M_k$ , and  $N_k$ , only admissible square-free values  $m$  are enumerated, so there is no SIMD filtering step of admissible values analogous to Section 7. However, as discussed in Section 3.6, these data structures have other drawbacks that make them a poor practical choice.

Staple’s recursive method is also not well suited for SIMD. It visits square-free values  $m$  in non-monotonic order and generates many small values  $m \leq m_{\min}$  that are discarded during the traversal, resulting in an irregular memory access pattern.

The precomputed square-free list is also not well suited for SIMD. Although it is stored contiguously, consecutive square-free values are irregularly spaced, which makes efficient vectorization impractical.

Among the approaches considered in Section 3, Oliveira e Silva’s dense lookup table and the factor table family are therefore the most natural SIMD-friendly representations for branchless filtering. In both cases, the hot loop scans a contiguous table in monotonic order and applies the same admissibility test to every loaded entry. Oliveira e Silva’s table is dense over all integers  $m$ , while the factor table is dense over the wheel-compressed set of represented integers. In both representations, consecutive table indexes correspond to increasing values of  $m$ , which makes vector loads, vector comparisons, and index compaction straightforward.

The crucial advantage of the factor table over Oliveira e Silva’s representation is its element size. Oliveira e Silva stores one full-size integer per represented value, whereas the factor table stores the same logical information in an integer type using only half as many bits. For example, a 16-bit factor table supports represented values up to about 32 bits, while a 32-bit factor table supports represented values up to about 64 bits. Thus, for a fixed SIMD register width, the factor table allows twice as many entries to be processed in each vector iteration as Oliveira e Silva’s data structure. This higher lane density is a key reason why the compact factor table is especially well suited for the branchless AVX512 and ARM SVE algorithms described in the following sections.

## 6 Branchless Scalar Algorithm for Hard Special Leaves

### 6.1 Baseline scalar loop

After the interval bounds have been converted into factor table indexes, the basic scalar algorithm scans the candidate range one index at a time. For each index  $m$ , it first tests whether the represented  $m$  value is admissible using the predicate `factor_table[m] > prime`. If the test succeeds, we calculate the contribution of the special leaf and add it to the sum.

The resulting scalar loop is:

```

m_min = factor_table.to_index(m_min);
m_max = factor_table.to_index(m_max);

for (int64_t m = m_max; m > m_min; m--)
{
    if (factor_table[m] > prime)
    {
        int64_t xpm = x / (prime * factor_table.to_number(m));
        int64_t count = sieve.count(xpm - low);
        int64_t phi_xpm = phi[b] + count;
        int64_t mu_m = (factor_table[m] & 1) ? -1 : 1;
        sum -= mu_m * phi_xpm;
    }
}

```

This loop is already more compact than the older versions based on separate lookup tables, but it still contains a data-dependent branch in the hot inner loop. In practice, this branch is highly unpredictable and causes many CPU branch mispredictions, which significantly deteriorate performance. Removing that branch is the goal of the next subsection.

## 6.2 Branchless filtering of admissible $m$

The key idea of the branchless algorithm is to split the scalar loop into two separate phases. The first phase performs only the filtering step and writes the admissible factor table indexes into a small stack array `m_indexes`. The second phase iterates over the filtered indexes and computes the corresponding special-leaf contributions. In conceptual form, the algorithm becomes:

```

int64_t m_count = 0;

for (int64_t m = m_max; m > m_min; m--)
{
    m_indexes[m_count] = m;
    m_count += (factor_table[m] > prime);
}

for (int64_t i = 0; i < m_count; i++)
{
    int64_t m = m_indexes[i];
    int64_t xpm = x / (prime * factor_table.to_number(m));
    int64_t count = sieve.count(xpm - low);
    int64_t phi_xpm = phi[b] + count;
    int64_t mu_m = (factor_table[m] & 1) ? -1 : 1;
    sum -= mu_m * phi_xpm;
}

```

In the actual `primecount` implementation [7], `m_indexes` is a small stack buffer of size 128, so the algorithm does not first filter the entire interval and only then process it. Instead, the scan over  $]m_{\min}, m_{\max}]$  repeatedly appends admissible indexes to `m_indexes`. Whenever the buffer becomes sufficiently full, the currently buffered leaves are processed immediately by iterating over `m_indexes`, after which the buffer is reset and the scan resumes. After the interval  $]m_{\min}, m_{\max}]$  has been exhausted, the remaining buffered indexes are processed once more. For the purpose of exposition, however, the simplified code above captures the essential idea.

The first loop is branchless because every candidate index  $m$  is written speculatively to the current output position `m_indexes[m_count]`. The comparison `factor_table[m] > prime` evaluates to either 0 or 1. If the condition is false, `m_count` is unchanged, so the just-written value is overwritten by the next iteration and is effectively discarded. If the condition is true, `m_count` is incremented by 1, so the written value becomes part of the filtered list. In this way, the unpredictable control-flow branch is replaced by a regular sequence consisting only of a store, a comparison, and an integer addition.

This transformation is important for two reasons. First, it eliminates the branch mispredictions discussed in Section 6.1. Second, it converts the filtering step into a compacting operation on a dense array of table indexes, which is exactly the form needed for the SIMD implementations described in Section 7.

## 7 SIMD Filtering of Admissible Values

### 7.1 SIMD design overview

The SIMD algorithm is not a different formula. It is the branchless filtering algorithm from Section 6.2 expressed in vector form. Instead of testing one factor table index  $m$  at a time, the SIMD code processes an entire vector of consecutive candidate indexes in each iteration. The corresponding factor table entries are loaded, compared against a vector whose lanes all contain the current prime, and the passing indexes are compacted into the same small buffer `m_indexes` that is used by the scalar buffered algorithm.

At a high level, both the ARM SVE and the AVX512 implementations follow the same sequence of steps:

1. Construct a vector of consecutive candidate indexes  $m, m - 1, m - 2, \dots$
2. Load the corresponding compressed factor table values.
3. Compare these values against a broadcast copy of `prime` in order to determine which candidates satisfy `factor_table[m] > prime`.
4. Compact the passing indexes and append them to `m_indexes`.
5. Once enough indexes have been collected, convert them back into integers, compute the corresponding values of  $x/(p \cdot m)$ , query the sieve counter, and accumulate the signed contributions.

Thus, the SIMD code keeps exactly the same high-level structure as the branchless scalar algorithm: first filter, then process. The essential difference is that the filtering phase now operates on many candidate indexes at once. This is possible because the factor table data is stored densely and monotonically, so a single vector load can fetch many adjacent entries and a single vector comparison can test all of them in parallel.

### 7.2 ARM SVE implementation

The ARM SVE implementation is a vector-length-agnostic realization of the branchless filtering algorithm from Section 6.2. Instead of hard-coding a specific vector width, it queries the number of available 32-bit lanes at runtime using `svcntw()` and then processes that many candidate indexes

per iteration. For simplicity, we only present the 32-bit code path, assume the common case in which the factor table uses `uint16_t` entries, and omit the tail loop.

The core ARM SVE loop is:

```
int64_t lanes32 = svcntw();
std::size_t m_count_max = m_indexes.size() - lanes32;
svbool_t all32 = svptrue_b32();
svuint32_t m_offsets32 = svindex_u32(0, 1);

for (; m >= m_min + lanes32; m -= lanes32)
{
    svuint32_t m_vec = svsub_u32_x(all32, svdup_n_u32(uint32_t(m)), m_offsets32);
    svuint32_t factor_vec = svrev_u32(svld1uh_u32(all32, &factor_table[m+1-lanes32]));
    svbool_t mask = svcmpgt_n_u32(all32, factor_vec, uint32_t(prime));
    int64_t matches = svcntp_b32(all32, mask);
    svuint32_t compact = svcompact_u32(mask, m_vec);
    svst1_u32(all32, &m_indexes[m_count], compact);
    m_count += matches;

    if (m_count > m_count_max)
    {
        for (std::size_t i = 0; i < m_count; i++)
        {
            int64_t m = m_indexes[i];
            int64_t xpm = x / (prime * factor_table.to_number(m));
            int64_t count = sieve.count(xpm - low);
            int64_t phi_xpm = phi[b] + count;
            int64_t mu_m = (factor_table[m] & 1) ? -1 : 1;
            sum -= mu_m * phi_xpm;
        }

        m_count = 0;
    }
}
```

This loop can be read from top to bottom as a direct vectorization of the scalar buffered algorithm. The vector `m_vec` contains the consecutive candidate indexes  $m, m-1, m-2, \dots$ . The instruction `svld1uh_u32()` loads `uint16_t` factor table entries and widens them to 32-bit lanes, and `svrev_u32()` reverses their order so that they align with `m_vec`. The comparison `svcmpgt_n_u32()` produces a predicate mask whose true lanes are exactly the admissible values satisfying `factor_table[m] > prime`.

The key instruction is `svcompact_u32()`. It packs only the passing candidate indexes to the front of the vector, thereby performing the same compaction step as the scalar write-pointer update from Section 6.2, but now on many candidates at once. The instruction `svcntp_b32()` counts how many lanes satisfied the predicate, and this count is then added to `m_count` in order to advance the output pointer into `m_indexes`.

Once the buffer is sufficiently full, the algorithm returns to the same processing phase as in the scalar version: convert the compact factor table indexes back into integers, compute the corresponding values of  $x/(p \cdot m)$ , query the sieve counter, and calculate the contribution of the special

leaf. Thus, the ARM SVE implementation does not change the mathematics of the algorithm; it accelerates only the filtering step by expressing it in a vector-length-agnostic SIMD form.

### 7.3 AVX512 implementation

The AVX512 implementation is a fixed-width realization of the branchless filtering algorithm from Section 6.2. For simplicity, we present only the 32-bit code path, which processes 16 candidate indexes per iteration, omit the tail loop, and do not inline the helper functions used for loading and reordering factor table values. Readers who want to inspect these helpers can find them in [D\\_avx512.hpp](#). The constant vectors used below are initialized before entering the loop.

The core AVX512 loop is:

```
constexpr std::size_t m_count_max = m_indexes.size() - 16;

for (; m >= m_min + 16; m -= 16)
{
    __m512i m_vec = _mm512_sub_epi32(_mm512_set1_epi32(uint32_t(m)), m_offsets32);
    __m512i factor_vec = load_factor_epi32_avx512(&factor_table[m - 15], reverse32);
    __mmask16 mask = _mm512_cmpgt_epu32_mask(factor_vec, prime_vec);
    _mm512_mask_compressstoreu_epi32(&m_indexes[m_count], mask, m_vec);
    m_count += popcnt64_native(mask);

    if (m_count > m_count_max)
    {
        for (std::size_t i = 0; i < m_count; i++)
        {
            int64_t m = m_indexes[i];
            int64_t xpm = x / (prime * factor_table.to_number(m));
            int64_t count = sieve.count(xpm - low);
            int64_t phi_xpm = phi[b] + count;
            int64_t mu_m = (factor_table[m] & 1) ? -1 : 1;
            sum -= mu_m * phi_xpm;
        }

        m_count = 0;
    }
}
```

As in the ARM SVE version, the vector `m_vec` contains the consecutive candidate indexes  $m, m-1, m-2, \dots$ . The helper `load_factor_epi32_avx512()` loads the corresponding compressed factor table entries into 32-bit lanes and reorders them so that they align with `m_vec`. The comparison `_mm512_cmpgt_epu32_mask()` then produces a 16-bit mask whose set bits correspond exactly to the admissible values satisfying `factor_table[m] > prime`.

The key AVX512 instruction is `_mm512_mask_compressstoreu_epi32()`. It writes only the passing candidate indexes to memory and packs them contiguously into `m_indexes`. The number of passing lanes is obtained using `popcnt64_native(mask)`, which advances the output pointer `m_count`. Thus, the AVX512 code performs the same compaction step as the scalar branchless algorithm and the ARM SVE implementation, but here the compaction is driven by an AVX512 mask and a compress-store instruction.

Once the buffer is sufficiently full, the algorithm returns to the same processing phase as in the scalar version: convert the compact factor table indexes back into integers, compute the corresponding values of  $x/(p \cdot m)$ , query the sieve counter, and calculate the contribution of the special leaf. Thus, the AVX512 implementation differs from the ARM SVE implementation mainly in the SIMD instruction set; algorithmically, both are vectorized realizations of the same branchless filtering scheme.

## 7.4 SIMD tail processing

In older SIMD implementations, the last few elements of a loop often had to be handled using a separate scalar cleanup loop because the remaining number of elements was smaller than the SIMD vector width. Modern SIMD instruction sets such as AVX512 and ARM SVE greatly simplify this situation. Both provide per-lane masking, so the final partial vector can still be processed using the same SIMD algorithm as a full vector.

The idea is simple. For the last block, one first constructs a mask or predicate that marks only the lanes corresponding to valid remaining elements. The load, comparison, compaction, and store operations are then executed only on these active lanes, while the inactive lanes are ignored. In AVX512 this is done using mask registers, whereas ARM SVE uses predicate registers. In both cases, the surviving candidate indexes are compacted exactly as in the full-width SIMD loop.

The main practical benefit is that no separate scalar tail loop is required. The same vectorized filtering algorithm can be used for both full blocks and the final partial block, which keeps the control flow simpler and avoids losing performance on the last few elements.

## 8 Increasing Instruction-Level Parallelism

The performance of the processing phase for hard special leaves of the form  $n = p \cdot m$  can be further improved by increasing instruction-level parallelism and reducing dependency chains. After the admissible values of  $m$  have been filtered, the algorithm evaluates the corresponding special leaf contributions. In this phase, the computation of

$$xpm = \frac{x}{p \cdot m}$$

introduces a noticeable amount of overhead. This is due to two factors. First, integer division is a high latency operation on modern CPUs. Second, the value of  $m$  is stored in compressed form and must be reconstructed using `factor_table.to_number(m)`, which introduces additional arithmetic overhead. In the straightforward implementation, the later operations in each iteration depend on the result of this computation. In particular, the sieve-count query and the update of the summation cannot proceed until the value of `xpm` has been computed. This limits instruction-level parallelism in the inner loop.

This limitation can be mitigated by restructuring the computation into two separate loops. In the first loop, the values of  $x/(p \cdot m)$  are computed for a batch of consecutive admissible values  $m$  and stored in a temporary array. In the second loop, these precomputed values are used to evaluate the corresponding contributions to the sum.

The key observation is that the division operations in the first loop are independent across iterations. This allows the CPU to overlap multiple divisions more effectively, thereby increasing instruction-level parallelism and improving overall throughput. In the second loop, the value of `xpm` can be read from the temporary cache array, which is significantly cheaper than recomputing the division and index-to-number conversion inside the hot loop.

The resulting structure is:

```
// Batch calculate x / (prime * m) to increase
// instruction level parallelism
for (std::size_t i = 0; i < m_count; i++)
    xpm_cache[i] = x / (prime * factor_table.to_number(m_indexes[i]));

// Main processing loop
for (std::size_t i = 0; i < m_count; i++)
{
    int64_t xpm = xpm_cache[i];
    int64_t count = sieve.count(xpm - low);
    int64_t phi_xpm = phi[b] + count;
    int64_t m = m_indexes[i];
    int64_t mu_m = (factor_table[m] & 1) ? -1 : 1;
    sum -= mu_m * phi_xpm;
}
```

In practice, this restructuring yields an additional speedup of up to 5% in the author’s benchmarks.

## 9 Benchmarks

In the benchmarks below, we compare the runtime of the new hard special leaves implementation in `primecount` 8.4 against the previous implementation in `primecount` 8.3, both using Gourdon’s D algorithm [3]. For each input value  $x$ , both versions were run several times, and the best runtime is reported. The test systems were a 16-core AMD EPYC 9R45 on AWS c8a.4xlarge running Fedora 43 and compiled with GCC 15.2; a 32-core AWS Graviton 4 on c8g.8xlarge running Fedora 43 and compiled with Clang 21; an Intel Core Ultra 5 245K with 14 CPU cores running Windows 11 and compiled with GCC 15.2 MinGW-w64; and an Apple M4 with 10 CPU cores running macOS Tahoe and compiled with Clang 22.

**AMD Zen5 9R45 - AVX512**

$x$	8.3 secs	8.4 secs	Speedup
$10^{17}$	0.341	0.268	1.27×
$10^{18}$	1.335	1.019	1.31×
$10^{19}$	5.186	3.986	1.30×
$10^{20}$	20.012	15.225	1.31×
$10^{21}$	77.573	58.844	1.32×
$10^{22}$	302.875	230.959	1.31×
$10^{23}$	1168.223	895.924	1.30×

**AWS Graviton 4 - ARM SVE**

$x$	8.3 secs	8.4 secs	Speedup
$10^{17}$	0.251	0.213	1.18×
$10^{18}$	0.961	0.820	1.17×
$10^{19}$	3.734	3.125	1.19×
$10^{20}$	14.571	12.321	1.18×
$10^{21}$	56.870	48.183	1.18×
$10^{22}$	226.277	194.079	1.17×
$10^{23}$	909.650	796.866	1.14×

Intel Core Ultra 5 245K - scalar

$x$	8.3 secs	8.4 secs	Speedup
$10^{17}$	0.491	0.398	$1.23\times$
$10^{18}$	1.886	1.536	$1.23\times$
$10^{19}$	7.272	5.993	$1.21\times$
$10^{20}$	28.704	23.717	$1.21\times$
$10^{21}$	111.966	91.776	$1.22\times$
$10^{22}$	444.437	363.416	$1.22\times$
$10^{23}$	1677.668	1350.387	$1.24\times$

Apple M4 - scalar

$x$	8.3 secs	8.4 secs	Speedup
$10^{17}$	0.907	0.797	$1.14\times$
$10^{18}$	3.444	3.024	$1.14\times$
$10^{19}$	13.855	11.747	$1.18\times$
$10^{20}$	53.149	45.100	$1.18\times$
$10^{21}$	204.443	174.006	$1.17\times$
$10^{22}$	794.797	680.861	$1.17\times$
$10^{23}$	3206.097	2737.922	$1.17\times$

These benchmarks show that the AMD Zen5 CPU benefits most from the new implementation, with speedups of up to 32%. This is consistent with its AVX512 implementation, which operates on 512-bit vectors. The AWS Graviton 4 with ARM SVE also sees consistent gains of about 14% to 19%. Even the Intel Core Ultra 5 245K and Apple M4 CPUs, which use the scalar branchless filtering path, still improve by up to 24% and 18%, respectively. This indicates that the benefit comes not only from SIMD vectorization, but also from removing unpredictable branches: on modern out-of-order processors, fewer branch mispredictions improve throughput and IPC significantly.

## 10 Conclusion

This paper has shown that the compact factor table provides a substantially more memory efficient representation of square-free numbers than the dense lookup tables used in earlier approaches. By packing square-freeness, Möbius sign, and least prime factor information into a single compact table entry, it preserves exactly the information needed for the admissibility test in the hard special leaves computation while also reducing memory usage and scan cost by a large constant factor. This same representation, in turn, enables a branchless scalar filtering algorithm and efficient SIMD implementations using AVX512 and ARM SVE instructions.

In the author’s earlier `primecount` implementations of the hard special leaves algorithm, many optimizations, such as pre-sieving with small primes, better thread scheduling, or improved integer division, typically yielded only modest single-digit speedups. By contrast, the branchless and SIMD filtering method developed here improves the performance of the hard special leaves algorithm by up to 32% in the author’s benchmarks. This makes it one of the most significant practical improvements to the hard special leaves algorithm in recent years and shows that the filtering step itself can be accelerated substantially when it is expressed using a compact factor representation and a branchless SIMD-friendly algorithm.

## References

- [1] J. C. Lagarias, V. S. Miller, and A. M. Odlyzko, *Computing  $\pi(x)$ : The Meissel-Lehmer method*, *Mathematics of Computation*, 44 (1985), pp. 537–560.
- [2] M. Deléglise and J. Rivat, *Computing  $\pi(x)$ : The Meissel, Lehmer, Lagarias, Miller, Odlyzko Method*, *Mathematics of Computation*, 65 (1996), pp. 235–245.
- [3] X. Gourdon, *Computation of  $\pi(x)$ : Improvements to the Meissel, Lehmer, Lagarias, Miller, Odlyzko, Deléglise and Rivat method*, February 15, 2001.



- [4] T. Oliveira e Silva, *Computing  $\pi(x)$ : The combinatorial method*, *Revista do DETUA*, 4(6) (2006), pp. 759–768.
- [5] D. B. Staple, *The combinatorial algorithm for computing  $\pi(x)$* , Master of Science thesis, Dalhousie University, Halifax, Nova Scotia, August 2015.
- [6] K. Walisch, *Efficient Computation of the Hard Special Leaves in the Combinatorial Prime Counting Algorithms*, June 13, 2025. Available at: <https://github.com/kimwalisch/primecount/blob/master/doc/Hard-Special-Leaves.pdf>
- [7] K. Walisch, *primecount: Fast C/C++ prime counting function library*, Version 8.4, 2026. Available at: <https://github.com/kimwalisch/primecount>
- [8] D. Ruiu, *fast-prime: A highly optimized prime counting toolkit in Rust*, Version 10, 2026. Available at: <https://github.com/secwest/fast-prime>