ORIGINAL ARTICLE

Converting an Integer to a Decimal String in Under Two Nanoseconds

Jaël Champagne Gareau[1] | **Daniel Lemire**[1]

[1]Université du Québec (TELUQ), Montreal, Quebec, H2S 3L5, Canada

Correspondence
Daniel Lemire, Université du Québec (TELUQ), Montreal, Quebec, H2S 3L5, Canada
Email: daniel.lemire@teluq.ca

Funding information
Natural Sciences and Engineering Research Council of Canada, Grant Number: RGPIN-2024-03787
Fonds de recherche du Québec,
https://doi.org/10.69777/361128

Converting binary integers to variable-length decimal strings is a fundamental operation in computing. Conventional fast approaches rely on recursive division and small lookup tables. We propose a SIMD-based algorithm that leverages integer multiply-add instructions available on recent AMD and Intel processors. Our method eliminates lookup tables entirely and computes multiple quotients and remainders in parallel. Additionally, we introduce a dual-variant design with dynamic selection that adapts to input characteristics: a branch-heavy variant optimized for homogeneous digit-length distributions and a branch-light variant for heterogeneous datasets. Our single-core algorithm consistently outperforms all competing methods across the full range of integer sizes, running 1.4–2× faster than the closest competitor and 2–4× faster than the C++ standard library function `std::to_chars` across tested workloads.

KEYWORDS
Integer numbers, String algorithms, Performance benchmarking

1 | INTRODUCTION

Converting binary integers into their decimal string representations is a fundamental operation in software systems. It appears in virtually all applications—whether for displaying values in command line or graphical user interfaces, writing diagnostic logs, or serializing data to text-based formats such as JSON, XML, or CSV. Despite its ubiquity, this conversion task has received relatively little attention in the formal peer-reviewed literature, and common implementations may leave room for performance improvements. As data formats and logging systems become performance bottle-

necks in modern applications, improving integer-to-string conversion speed directly benefits real-world software such as database engines, serialization libraries, and logging frameworks.

Modern processors provide several architectural features that could potentially be exploited to improve the performance of such conversions. SIMD (Single Instruction, Multiple Data) extensions, for example, allow the same arithmetic operation to be applied to multiple data elements in parallel. On recent x86-64 processors from Intel and AMD, the AVX-512 instruction sets can process 512-bit registers using single instructions, offering potential for accelerating numerical workloads [1, 2]. At the same time, processors can execute multiple independent instructions simultaneously (*instruction-level parallelism*) including load and store operations. To improve such parallelism, contemporary out-of-order execution engines can reorder instructions and use speculative execution to hide latencies and improve throughput. However, speculative execution might suffer from branch mispredictions. While branch predictors have become increasingly sophisticated, mispredicted branches still introduce performance penalties. These architectural factors interact in subtle ways when implementing high-performance integer-to-string conversions. Achieving both low latency and high throughput requires careful coordination between algorithmic structure and processor microarchitecture, including vector width utilization, instruction scheduling, and branch prediction behavior.

Our *primary contribution* is a new integer-to-string conversion algorithm that explicitly leverages SIMD parallelism. Our design exploits the AVX-512 instruction set to process multiple digits concurrently, computing quotients and remainders in parallel without costly integer divisions. Furthermore, a distinctive feature of our approach is its dual-variant mechanism, which dynamically selects between two specialized conversion paths based on sampled input characteristics: a branch-heavy variant optimized for homogeneous digit-length distributions and a branch-light variant for heterogeneous datasets. This adaptive design balances branch prediction accuracy with instruction-level parallelism.

As a *secondary contribution*, we present, to the best of our knowledge, the first formal empirical study of integer-to-string conversion performance on a modern AVX-512-capable processor. Our evaluation combines both randomized datasets—with controllable digit-length distributions—and data extracted from real-world files. In particular, we analyze how the distribution of digit lengths within the input affects throughput, revealing that this factor alone dominates overall performance. We compare our approach against widely used library implementations, and our results highlight where each variant excels and how architectural features influence observed performance.

The remainder of this paper is organized as follows. Section 2 reviews prior work and commonly used algorithms, including those found in standard libraries. Section 3 provides a brief overview of SIMD instructions and the AVX-512 extension relevant to our algorithm. Section 4 presents the mathematical foundation for our approach. Section 5 describes the design of our proposed algorithm, with emphasis on its SIMD data layout, parallel digit extraction, and dynamic variant-selection mechanism. Section 6 describes the benchmarking methodology, hardware platform, and performance metrics. Finally, Section 7 discusses the implications of our results and outlines possible extensions.

2 | RELATED WORK

Formally, the problem consists of converting an integer n into its decimal representation using the characters −, 0, . . ., and 9. In Unicode and ASCII, the digit characters have code point values ranging from 48 (for 0) to 57 (for 9) consecutively. Thus, to convert an integer value k between 0 and 9 to a string, it suffices to generate the character with the code point value '0' + k, where '0' is a shorthand for the code point value 48. In the general case, and ignoring negative values, we seek a sequence of characters $c_0 c_1 \dots c_{k-1}$ such that $n = \sum_{i=0}^{k-1} (c_i - \text{'0'}) \cdot 10^{k-1-i}$, where k is the number of digits in the representation of n.[1] In practice, most algorithms first check whether n is negative; in

[1]The number of decimal digits needed to represent a positive integer n is $\lfloor \log_{10} n \rfloor + 1$.

which case, they prepend a minus sign ('-') and proceed with the absolute value. For simplicity, we focus principally on unsigned integers and, more precisely, on 64-bit strictly positive values (i.e., $n \in [1, 2^{64} - 1]$).

```
1  size_t uint64_to_str_naive(uint64_t n, char *buffer) {
2      char *p = buffer;
3      while (n > 0) {
4          const uint64_t digit = n % 10;
5          n /= 10;
6          *p++ = '0' + digit;
7      }
8      std::reverse(buffer, p);
9      return p - buffer; // Return length
10 }
```

FIGURE 1 C++ implementation of the classic integer-to-string conversion algorithm [3].

Naive and Baseline Methods.
The classic approach repeatedly divides and takes the modulus by 10 to extract digits from least to most significant, as illustrated in Figure 1. Because digits are produced in reverse order, a final reversal step is required [3]. Some variants instead write the number right-to-left into a buffer and then copy the resulting string to its correct position. More advanced implementations precompute the number of digits beforehand [4] to place each character directly at its final position, avoiding both reversal and data movement, thereby enabling a single-pass implementation.

Two-Digit Lookup Methods.
A common optimization is to reduce the number of division and modulus operations. Instead of extracting one digit per iteration, pairs of digits are produced by dividing by 100 and mapping the result (0–99) through a small lookup table of two-character strings. The final odd digit, if any, is handled separately. This strategy was popularized by Alexandrescu [5] and adopted in several high-performance formatting libraries, such as `fmt`.[2] It removes many divisions and relies on cache-friendly, sequential memory accesses, which are efficiently handled by modern CPUs.

Division by Constants.
The conversion of integers to decimal digits often involves divisions by 10 or powers of two. Division instructions on mainstream computers are typically slower than multiplications.[3] For this reason, optimizing compilers replace divisions by *multiplicative divisions*: a multiplication followed by a shift [7, 8, 9, 10, 11, 12, 13]. The algorithm used by optimizing compilers usually follows Warren's approach [14]. We can concisely summarize the mathematical results with optimal bounds [15]. Define $\text{division}(x, y) \coloneqq \text{floor}(x/y)$ and $\text{remainder}(x, y) \coloneqq x - \text{division}(x, y) \cdot y$ for positive real numbers x, y with the constraint that $y \neq 0$. Given a non-zero divisor d and a non-negative numerator, we have that $\text{division}(n, d) = \text{division}(c \cdot n, m)$ for all $n \in [0, N]$ if $1/d \leq c/m < \left(1 + \frac{1}{N - \text{remainder}(N+1, d)}\right) 1/d$, where c and m are non-negative integers. We can require that $c \in [0, m)$. By choosing m to be a power of two, the formula

[2] `https://github.com/fmtlib/fmt/blob/9395ef5fcb81/include/fmt/format.h#L1219`

[3] On recent AMD processors with a Zen 5 microarchitecture, a division instruction might require between 11 and 19 cycles while a multiplication requires only 3 cycles [6].

division(n, d) = division$(c \cdot n, m)$ indicates that we can replace a division by an arbitrary positive integer d by a multiplication $(c \times d)$ followed by a shift (division by m). Fig. 2 illustrates how we might compute c in practice.

```
def find(N, d):
    L = 1
    while True:
        m = 2 ** L
        c = (m + d - 1) // d
        ra = N - (N + 1) % d
        if c * d * ra < m * (ra + 1):
            return L, c
        L += 1
```

FIGURE 2 Python script to compute the constant for a multiplicative division. It finds L and c such that $\lfloor n/d \rfloor = \lfloor (c \times n)/2^L \rfloor$ for all $n \in [0, N]$.

Large-Chunk Precomputed Methods.

The same idea can be extended to larger digit groups, most notably 4-digit and 5-digit blocks. However, table size grows exponentially with block width (40 KB for a 4-digit table), often exceeding the L1 data cache. Consequently, such approaches trade arithmetic reduction for cache locality. Examples include Chateauneu's 4-digit `itoa` implementation [16] and Henriksen's 5-digit variant [17] derived from the GNU `roff` project's `itoa` implementation. The `fmt` contributors later implemented a more aggressive multi-table algorithm originally proposed by user "u2985907" [18], which heavily unrolls loops and merges large precomputed chunks to minimize branches and divisions—trading instruction count for instruction-cache pressure. Hopman's "`hopman_fast`" algorithm [19] uses a similar 4-digit strategy but attaches a small "leading-zero" flag to each chunk, allowing zero-padding to be skipped quickly when combining ASCII characters. For example, for the number 112 345 678, you get the three chunks 0001, 1234, 5678 corresponding to $n/10^8$, $(n/10^4) \bmod 10^4$, and $n \bmod 10^4$ where $n = 112345678$. You end up with many leading zero bytes in the higher chunks that must not appear in the final string. Hopman's solution embeds the number of leading zeroes inside the highest byte of the 4-byte chunk. In contrast, Henriksen's version operates on 5-digit blocks and maintains separate forward and reverse tables of strings. The forward tables contain the strings corresponding to integers 0 to $10^5 - 1$ inclusively, and from $10^5 - 1$ to 0 inclusively for the reverse tables. The forward table is used with positive integers: we lookup the string starting from the beginning of the table. The reverse table is used for negative integers: starting from the end of the table, we use a negative index in the table to lookup the string.

Arithmetic and Fixed-Point Methods.

Mathisen proposed a fully arithmetic approach that avoids lookup tables altogether [20]. The method partitions a 32-bit integer into two five-digit segments, processed concurrently by exploiting instruction-level parallelism. Each sub-conversion uses a 4.28 fixed-point representation (4 bits integer, 28 bits fractional) in which digits are extracted via bit shifts and successive multiplications by ten. A small bias term corrects rounding errors in the reciprocal of 10 000. By interleaving the two halves and unrolling the loop, Mathisen's method achieved sub-hundred-cycle conversion times on late-1990s CPUs without divisions or tables. A later Usenet post [21] by Mathisen mentioned the possibility of a 64-bit extension but did not provide a working implementation.

Building on similar fixed-point principles, Hopman's "fun" algorithm [19] follows a related philosophy: it recursively splits the number into 8-, 4-, and 2-digit blocks using reciprocal-multiplication constants approximating division by powers of ten. Each stage decomposes the value into higher and lower blocks until every block represents a single digit. Afterward, all digits are converted to ASCII in bulk via a single *bitwise OR* with `0x30`, and leading zeros are skipped branchlessly. This design also exploits instruction-level parallelism and fixed-point arithmetic to achieve high throughput without tables or divisions.

The AppNexus Common Framework's implementation[4] of integer-to-string conversion [22] follows a conceptually similar strategy. It recursively splits the number into base-10^k chunks (with $k \in \{4, 8\}$) using fixed-point reciprocals to compute both quotient and remainder without divisions.

A related production-grade implementation is Abseil's `FastIntToBuffer` routine.[5] For 32- and 64-bit integers, it decomposes the value into blocks of up to eight digits, then uses fixed-point reciprocals of 10, 10^2, and 10^4 to compute decimal digits in parallel inside 32- or 64-bit registers. The resulting bytes are turned into ASCII by adding packed "zero" patterns and storing them with unaligned 16/32/64-bit writes, so no per-digit or per-block digit table is needed.

Hybrid Approaches.

Several recent integer-to-string conversion routines combine elements from both the lookup-table and arithmetic/fixed-point families, yielding hybrid methods that avoid costly divisions while keeping table sizes small and cache-friendly.

A representative example is `jeaiii`'s algorithm,[6] which decomposes the input integer into blocks whose sizes depend on the value range. Each block is converted using a sequence of reciprocal-multiplication steps based on fixed-point constants, followed by two-digit lookups into a compact 200-byte table. The algorithm aggressively unrolls these operations and relies on a mixture of masked multiplications, shifts, and lookup pairs to extract multiple digits at once.

A further example of this hybrid style is Yaoyuan's `itoa_yy` implementation.[7] It uses a compact 200-byte lookup table for all two-digit combinations (00–99), while performing all higher-level decompositions purely via fixed-point arithmetic. For 32-bit integers, the algorithm selects specialized code paths for 1–2, 3–4, 5–6, 7–8, and 9–10 digits, each using carefully chosen constants to implement divisions by 10, 10^2, 10^4, and 10^8 through multiplication and bit shifts. The 64-bit path recursively splits the input into 4- and 8-digit blocks, reusing the same two-digit table to materialize all decimal pairs. There is also a "large lookup-table" variant that uses 50 kB of tables. Both variants are shown to be among the fastest scalar routines reported in informal benchmarks.[8]

SIMD Methods.

The emergence of SIMD instruction sets has enabled digit extraction to be performed on multiple integers in parallel. Muła [23] demonstrated two SSE2-based routines: the first (originally designed by Wyderski and adapted by Muła) converts two 8-digit 32-bit integers concurrently, while the second processes a single integer with lower latency. Both use reciprocal multiplication and vector arithmetic but are limited to 32-bit inputs.

Later experiments by the StackOverflow user "icecreamsword" [24] attempted to vectorize Mathisen's algorithm using SSE4.1 by splitting numbers into uniform blocks (e.g., three groups of four digits). The SIMD version achieved roughly a 20–30% improvement (34–42 cycles vs. 44–50 cycles for the scalar assembly version), demonstrating modest but consistent benefits from parallelization.

[4] `https://github.com/appnexus/acf/blob/5a4645d/src/an_itoa.c`

[5] `https://github.com/abseil/abseil-cpp/blob/d97663e/absl/strings/numbers.h#L204`

[6] `https://github.com/jeaiii/itoa/blob/c861d1c/include/itoa/jeaiii_to_text.h`

[7] `https://github.com/ibireme/c_numconv_benchmark/blob/8541662/src/itoa/itoa_yy.c`

[8] `https://github.com/ibireme/c_numconv_benchmark`

Implementations in C++ Standard Libraries.
The C++17 function `std::to_chars` provides a standardized low-level integer-to-string conversion routine whose implementations differ across libraries and may evolve over time. At the time of writing, the GNU `libstdc++` version adopts the same two-digit-table strategy described above.[9] LLVM's `libc++` implementation likewise employs a two-digit lookup table but, for 64-bit values, divides the number by 10^{10}: it formats the quotient ($< 2^{64}/10^{10} < 2^{31}$) when it is non-zero using the 32-bit routine and then appends the low ten digits through an `append10` helper (which computes ten digits). This hierarchical reuse of the smaller routine yields a branch-light and uniform inner loop.[10] By contrast, Microsoft's `MSVC` implementation retains the classic repeated-division method shown in Figure 1.[11] The source code explicitly notes that "Ryū's digit table should be faster here," referring to the lookup table used in the Ryū float-to-string conversion algorithm [25], which is essentially the same two-digit table employed by the two-digit lookup approaches discussed earlier.[12]

3 | SIMD INSTRUCTIONS AND AVX-512

Modern processors provide SIMD extensions that apply the same operation to several data elements packed within wide registers. By contrast with scalar instructions, which process one value per operation, SIMD instructions exploit data-level parallelism. Almost all commodity processors today support some form of SIMD instructions. A SIMD register spanning, say, 64 bytes, can represent 64 bytes, 32 shorts (16-bit words), 16 integers (32-bit words), or 8 long integers (64-bit words). Similarly, floating-point numbers are supported.

On the x86-64 architecture, SIMD has evolved through several generations. SSE and SSE2 (128-bit vectors) provided the initial foundation for packed integer and floating-point arithmetic. AVX (2008) and AVX2 (2013) doubled the register width to 256 bits and added support for fused multiply-add (FMA) operations as well as richer integer and gather/scatter instructions. AVX-512, introduced by Intel starting with the Knights Landing Xeon Phi (2016) and later extended to mainstream Core and Xeon processors (Skylake-X, Ice Lake, Sapphire Rapids, Granite Rapids, and equivalents from AMD starting with Zen 4), further widens the vectors to 512 bits and increases the number of architectural vector registers from 16 to 32. AVX-512 also introduces an EVEX prefix that enables operation masking with dedicated mask registers. It becomes possible, for example, to write, load or multiply only some of the elements of a SIMD registers. A mask is conceptually just a word made of 8 bits, 16 bits, 32 bits or 64 bits—corresponding to the number of elements represented in the register.

AVX-512 comprises multiple optional subsets, not all of which are supported on every processor that implements AVX-512. The foundational AVX-512F subset provides the core 512-bit floating-point and integer arithmetic. Other important extensions include AVX-512VL (vector-length orthogonality, allowing most instructions to operate on 128-bit or 256-bit vectors), AVX-512DQ/BW (32/64-bit and 8/16-bit integer operations), and AVX-512CD (conflict detection for sparse gather operations). The subset most relevant to our work is AVX-512IFMA (Integer Fused Multiply-Add), which adds the instructions `vpmadd52luq` and `vpmadd52huq`. These instructions perform a 52-bit $\times$ 52-bit unsigned multiplication on each of the eight 64-bit lanes of a 512-bit register, producing a 104-bit product per lane. The `vpmadd52luq` variant writes the low 52 bits of each product (plus an addend) back to the destination, while the `vpmadd52huq` variant writes the high 52 bits. When programming, we can invoke these instructions using *intrinsics* such as `_mm512_madd52lo_epu64` and `_mm512_madd52hi_epu64`. SIMD intrinsics are special functions giving

[9] https://gcc.gnu.org/onlinedocs/gcc-15.2.0/libstdc++/api/a00644_source.html#l00083
[10] https://github.com/llvm/llvm-project/blob/c2b2a347bf94/libcxx/include/__charconv/to_chars_base_10.h
[11] https://github.com/microsoft/STL/blob/38d7248/stl/inc/charconv
[12] https://github.com/ulfjack/ryu/blob/023ee6b/ryu/digit_table.h

the programmer access to SIMD instructions. AVX-512IFMA is available on AMD processors starting with the Zen 4 microarchitecture (2022) and on Intel server processors starting with the Ice Lake microarchitecture (2019).

The 52-bit precision matches well with decimal computations. For example, to compute a high-accuracy approximation of $n/10^k$ or $n \bmod 10^k$ for $k \leq 8$, one can multiply n by a carefully chosen reciprocal constant near $2^{52}/10^k$ and then extract either the high or low half of the product. The fused nature of the instruction (multiply + add in one operation) reduces latency and instruction count compared with separate multiply and add steps. Furthermore, the `vpmadd52luq` and `vpmadd52huq` instructions are relatively inexpensive: on recent AMD processors with a Zen 5 microarchitecture, they execute in only four cycles, and two can be retired per cycle [6]. These properties make AVX-512IFMA particularly attractive for fast decimal digit extraction without lookup tables or scalar divisions.

4 | MATHEMATICAL RESULT

Lemire et al. [26] introduced a fast method for computing remainders using multiplicative inverses. This approach allows us to compute the remainder of an integer division by a power of ten using only multiplications and bit shifts We adapt the result to our specific needs and provide a proof of the following theorem, which is a key component of our algorithm. We state the main result after recalling the following preliminary lemma from Lemire et al. [15]. Recall that we define $\mathrm{division}(x, y) \coloneqq \mathrm{floor}(x/y)$ and $\mathrm{remainder}(x, y) \coloneqq x - \mathrm{division}(x, y) \cdot y$ for positive real numbers x, y with $y \neq 0$.

Lemma 1 *Consider a positive integer $d > 0$, a non-negative integer n, and a non-negative real number x. We have* $\mathrm{remainder}(n, d) = \mathrm{floor}(\mathrm{remainder}(x, d))$ *and* $\mathrm{division}(n, d) = \mathrm{division}(x, d)$ *if and only if $n \leq x < n + 1$.*

By substituting a power of two for m and 10 for d, the following theorem gives us a formula to compute any digit of the integer n with only two multiplications—when not counting multiplications by a power of two as they can be implemented as a binary shift.

Theorem 2 *Consider a base $d \geq 2$, an exponent $k \geq 1$ (e.g., $d = 10$ and $k \in [1, 8]$ for decimal digit extraction), an upper bound N, and positive integers c and m. We have* $\mathrm{remainder}(\mathrm{division}(n, d^{k-1}), d) = \mathrm{division}(\mathrm{remainder}(c \cdot n + c, m) \cdot d, m)$ *for all integers $n \in [0, N]$ if*

$$\left(1 - \frac{1}{N+1}\right) \frac{1}{d^k} \leq \frac{c}{m} < \frac{1}{d^k}.$$

Proof Let $p = d^{k-1}$ and $D = d^k = d \cdot p$. For each $n \in [0, N]$, let $q = \mathrm{division}(n, p) = \mathrm{floor}(n/p)$. The goal is to prove

$$\mathrm{remainder}(q, d) = \mathrm{division}\big(\mathrm{remainder}(c \cdot n + c, m) \cdot d, m\big).$$

We simplify the right-hand side. By definition of remainder,

$$\mathrm{remainder}(c \cdot n + c, m) = c(n + 1) - \mathrm{division}(c(n + 1), m) \cdot m.$$

Multiplying both sides by d/m gives

$$\mathrm{remainder}(c \cdot n + c, m) \cdot \frac{d}{m} = \frac{c(n+1)d}{m} - \mathrm{division}(c(n + 1), m) \cdot d.$$

Let $x = c(n+1)d/m$. The expression above becomes $x - \text{division}(c(n+1), m) \cdot d$. Since $x/d = c(n+1)/m$, we have $\text{division}(c(n+1), m) = \text{floor}(x/d)$, and thus $x - \text{division}(c(n+1), m) \cdot d = \text{remainder}(x, d)$. Therefore,

$$\text{division}\big(\text{remainder}(c \cdot n + c, m) \cdot d, m\big) = \text{floor}\big(\text{remainder}(x, d)\big).$$

Thus we only need to show that $\text{remainder}(q, d) = \text{floor}\big(\text{remainder}(x, d)\big)$ for every $n \in [0, N]$. By Lemma 1 (applied with numerator q and real number x), the two statements

$$\text{remainder}(q, d) = \text{floor}\big(\text{remainder}(x, d)\big) \qquad \text{and} \qquad \text{division}(q, d) = \text{division}(x, d)$$

hold simultaneously if and only if $q \le x < q + 1$. In particular, the first equality (the one we need) holds when $q \le c(n+1)d/m < q + 1$, or equivalently

$$\frac{q}{d(n+1)} \le \frac{c}{m} < \frac{q+1}{d(n+1)}.$$

It therefore suffices to show that the given interval for c/m,

$$\left(1 - \frac{1}{N+1}\right)\frac{1}{D} \le \frac{c}{m} < \frac{1}{D},$$

lies inside the required interval for every $n \in [0, N]$. This is equivalent to showing two uniform bounds:

1. (Upper bound.) For every $n \in [0, N]$, we need to show that

$$\frac{q+1}{d(n+1)} \ge \frac{1}{D}.$$

Equivalently, $(q+1)/(n+1) \ge 1/p$. Write $n = qp + r'$ with $0 \le r' < p$ (so $q = \text{floor}(n/p)$). Then $n + 1 \le (q+1)p$, and therefore

$$\frac{q+1}{n+1} \ge \frac{q+1}{(q+1)p} = \frac{1}{p},$$

with equality precisely when the block for this q reaches its natural end $n = (q+1)p - 1$. (If the last block for $q_{\max} = \text{floor}(N/p)$ is incomplete, the inequality is strict.) Multiplying by $1/d$ yields the claimed lower bound on the required upper bound for c/m.

2. (Lower bound.) For every $n \in [0, N]$, we need to show that

$$\frac{q}{d(n+1)} \le \left(1 - \frac{1}{N+1}\right)\frac{1}{D} = \frac{N}{dp(N+1)}.$$

Equivalently,

$$\frac{q}{n+1} \le \frac{N}{p(N+1)}.$$

Inside each block where q is constant, the left-hand side is maximized at the smallest n of the block, i.e., at $n = qp$

(where $n+1 = qp+1$). Thus, it is enough to bound the values

$$\frac{q}{qp+1} = \frac{t/p}{t+1} = \frac{t}{p(t+1)}, \quad t = qp.$$

The function $q \to q/(q+1)$ is strictly increasing for $q \geq 0$. The global maximum occurs in the last block: let $q_{\max} = \mathrm{floor}(N/p)$, $r = N \bmod p$, so $t = q_{\max}p = N - r$ with $0 \leq r < p$. Then the maximum value attained is

$$\frac{t}{p(t+1)} = \frac{N-r}{p(N-r+1)}.$$

Because $t = N - r \leq N$, we have

$$\frac{t}{t+1} \leq \frac{N}{N+1} \implies \frac{t}{p(t+1)} \leq \frac{N}{p(N+1)}.$$

Consequently every required lower bound on c/m is at most the left endpoint of the given interval.

Because the given interval for c/m satisfies

$$\max_n \frac{q}{d(n+1)} \leq \left(1 - \frac{1}{N+1}\right)\frac{1}{D} \leq \frac{c}{m} < \frac{1}{D} \leq \min_n \frac{q+1}{d(n+1)},$$

the inequality $q \leq x < q+1$ holds for every $n \in [0, N]$. By the one-sided implication of Lemma 1 we obtain the desired remainder identity. This completes the proof.

5 | METHODS

This section presents our proposed SIMD integer-to-string conversion algorithm. We begin with an overview of the overall design, followed by detailed descriptions of the SIMD kernel, multiplicative division technique, and dynamic variant-selection mechanism.

5.1 | Overview

Our approach decomposes 64-bit integers into two 8-digit chunks by dividing the integers by 10^8: $n/10^8$ and $n \bmod 10^8$. Optimizing compilers typically convert a division by 10^8 into a multiplication by $\lceil 2^{90}/10^8 \rceil$ followed by a right shift. The remainder can be computed by a multiplication of the quotient by 10^8 followed by a subtraction. Our approach then processes each chunk in parallel using AVX-512 vector instructions to extract all eight decimal digits simultaneously, and emits the results via masked or fixed-width stores. The core innovation is a vectorized kernel (§ 5.2) that replaces traditional repeated divisions with parallel multiplicative remainders computed using AVX-512 IFMA instructions. We build two complete conversion routines on this kernel: a heterogeneous variant (§ 5.4) optimized for mixed digit lengths, and a homogeneous variant (§ 5.5) specialized for uniform-length batches. A lightweight profiling step (§ 5.6) automatically selects the best variant for each dataset, balancing branch prediction accuracy and instruction-level parallelism.

5.2 | SIMD Algorithm: Parallel Computation of Decimal Remainders and ASCII Packing

The core of our algorithm is a vectorized kernel that converts a single integer (up to 16 digits) into its ASCII decimal representation using a small number of SIMD instructions. Rather than extracting digits one or two at a time through repeated division, we compute all remainders $n \bmod 10^k$ for $k = 1, \ldots, 8$ in parallel within a single 512-bit register, then convert them to ASCII digits in bulk. This parallelism is enabled by the Integer-Fused-Multiply-Add (IFMA) instructions available in the AVX-512 instruction set extension.[13] The IFMA instructions allow us to compute multiplications and additions on 52-bit limbs efficiently (each 64-bit lane carries a 52-bit multiplicand), which is ideal for our purpose of extracting decimal digits.

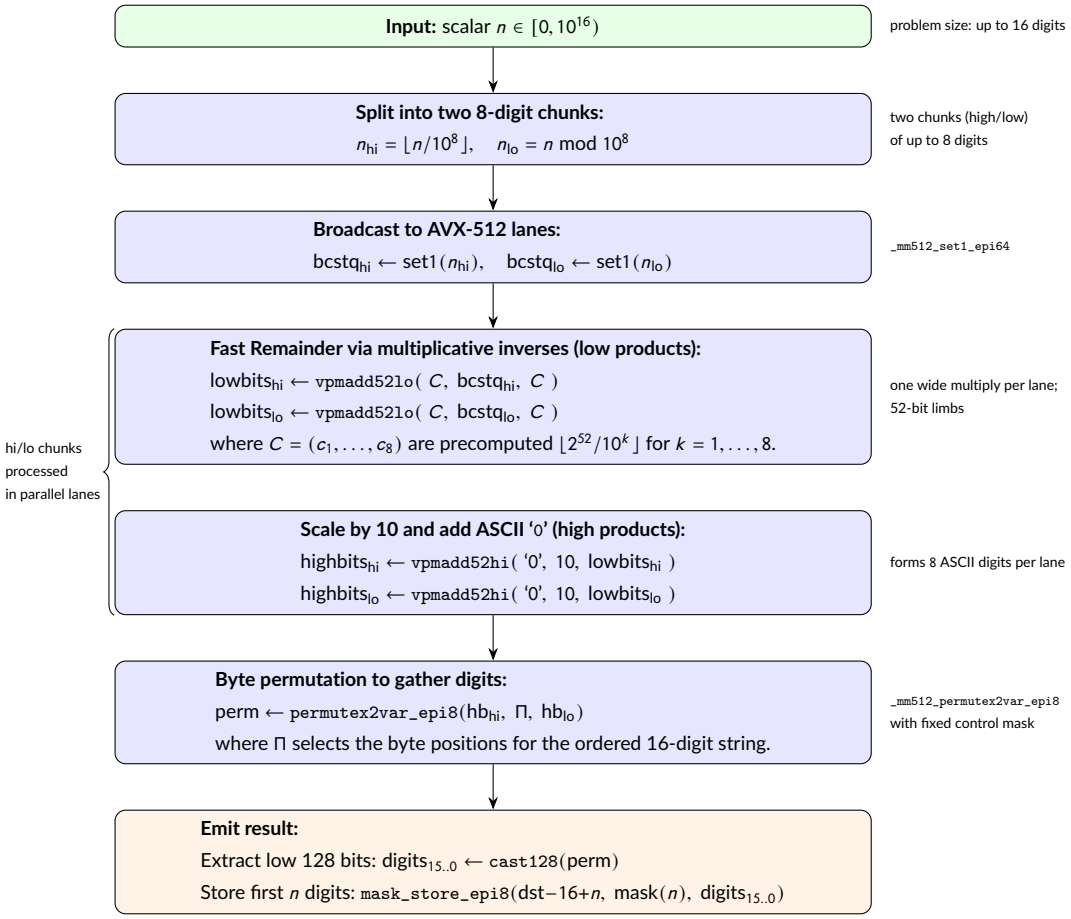

FIGURE 3 SIMD algorithm for integer-to-string split into two 8-digit chunks, lane-parallel fastmod via `vpmadd52`, byte permutation to assemble ordered digits, and masked store to the output buffer.

At its core, the kernel computes all remainders (aka residues) $n \bmod 10^k$ for $k = 1, \ldots, 8$ *in parallel* and turns them into ASCII digits. In more detail, our kernel first splits the input integer n—in $[0, 10^{16})$—into two 8-digit chunks

[13] https://en.wikichip.org/wiki/x86/avx512_ifma

```
__m512i to_string_8digits(uint64_t n) {
  __m512i vn = _mm512_set1_epi64(n);
  __m512i c = _mm512_setr_epi64(
   45035996, 450359962, 4503599627, 45035996273,
   450359962737, 4503599627370, 45035996273704,
   450359962737049);
  __m512i vten = _mm512_set1_epi64(10);
  __m512i vzero = _mm512_set1_epi64('0');
  __m512i low = _mm512_madd52lo_epu64(c, vn, c);
  return _mm512_madd52hi_epu64(vzero, vten, low);
}
```

(a) C function to compute the eight ASCII digits of $n < 10^8$

```
to_string_8digits(unsigned long):
 vmovdqa64 zmm1, zmmword ptr [rip+.L0]
 vpbroadcastq zmm0, rdi
 vpmadd52luq zmm1, zmm0, zmm1
 vpbroadcastq zmm0, qword ptr [rip+.L1]
 vpmadd52huq zmm0, zmm1, qword ptr[rip+.L2]{1to8}
 ret
```

(b) Equivalent assembly

FIGURE 4 Side-by-side comparison of a C function to compute eight digits and its assembly equivalent

(high/low). Each chunk is then broadcast across eight lanes and multiplied by precomputed constants $c_k = \lceil 2^{52}/10^k \rceil$ (see § 5.3). A second fused multiply-add is then used to scale by 10 and add the ASCII offset '0'. Concretely, we use `VPMADD52LUQ` to form "low" 52-bit products (one per lane), followed by `VPMADD52HUQ` to compute $10 \cdot \text{low} + \text{'0'}$. A single byte-wise lane crossbar (`permutex2var_epi8`) then gathers the eight digits of both 8-digit chunks together. Finally, a masked store can write only the significant prefix to the output buffer when the input n has fewer than 16 digits. Figure 3 shows the various stages of the computation.

5.3 | Fast Remainder via Multiplicative Inverses

The SIMD algorithm from § 5.2 instantiates the fast computation of decimal remainders based on Theorem 2 to compute the multiplicative inverses needed for our SIMD kernel. We have that $\text{remainder}(\text{division}(n, d^{k-1}), d) = \text{division}(\text{remainder}(c \cdot n + c, m) \cdot d, m)$ for all $n \in [0, N]$ if $\left(1 - \frac{1}{N+1}\right) 1/d^k \leq c/m < 1/d^k$. Setting $m = 2^{52}$ and $N = 10^8 - 1$, we can compute the constant c as $\lfloor 2^{52}/d^k \rfloor$. We can verify that for $d = 10^k$ with $k \in [1, 8]$, the constant c satisfies the required bounds, ensuring that the formula holds for all $n \in [0, 10^8)$. See Figure 5 for a Python script that verifies these bounds for the relevant powers of ten. Specifically, we have that $\text{remainder}(\text{division}(n, 10^{k-1}), 10) = \text{division}(\text{remainder}(c \cdot n + c, m) \cdot 10, m)$ for all $n \in [0, 10^8)$ and $k \in [1, 8]$. In practice, we precompute these constants $\lfloor 2^{52}/d^k \rfloor$ for eight powers of ten and use them in our SIMD kernel to compute the remainders efficiently. The first step $\text{remainder}(c \cdot n + c, m)$ is computed using the `vpmadd52lo` instruction, which performs the multiplication and addition in one step, while the second step $\text{division}(\text{remainder}(c \cdot n + c, m) \cdot d, m)$ is computed using the `vpmadd52hi` instruction, which scales by 10 and extracts the high 52 bits to yield the final digit values.

Figure 4 illustrates our main routine with a code snippet and its equivalent assembly. It converts an integer $n < 10^8$ into its eight-digit ASCII string representation using AVX-512 instructions. The C function broadcasts the input n across eight 64-bit lanes in the vector `vn`. It then utilizes a vector `c` containing precomputed constants approximating $\lfloor 2^{52}/10^k \rfloor$ for $k = 8, \ldots, 1$. The computation of `low` via `_mm512_madd52lo_epu64(c, vn, c)` adds the original constants to the low 52 bits of the products $n \cdot c$. Finally, `_mm512_madd52hi_epu64(vzero, vten, low)` adds the ASCII value of "0" (broadcast in `vzero`) to the high 52 bits of $10 \cdot$`low`, producing the character codes for each digit in the respective lanes of the returned `__m512i`. The accompanying assembly code is the compiled equivalent, loading

```python
for k in range(1, 9):
  d = 10 ** k
  lhs = (10**8 - 1) * 2**52
  rhs = (2**52 // d) * d * 10**8
  assert lhs <= rhs
```

FIGURE 5 Python script to verify that the constants $\lfloor 2^{52}/d^k \rfloor$ satisfy the bounds required by Theorem 2 for $d = 10^k$ and $N = 10^8 - 1$.

the constant vector `c` into `zmm1` from memory, broadcasting n into `zmm0`, computing the low fused multiply-add, then setting `zmm0` to a broadcast of "0" from `.L1`, and performing the high fused multiply-add with a broadcast of 10 from `.L2`, resulting in the digit characters ready for packing or storage. Thus, not counting the return instruction, we require only five instructions to compute the eight-digit ASCII string.

5.4 | Base Conversion Routine

Building upon the SIMD algorithm from § 5.2, we implement the complete integer-to-string conversion in the function `avx512_to_chars`. This routine converts a 64-bit unsigned integer into its decimal representation using one or two vectorized 8-digit blocks and a scalar fallback for the final four digits. It relies on masked vector stores to handle variable-length numbers without branching. Figure 6 presents a representative implementation of this conversion routine in C++.

The algorithm first determines the number of digits using a fast branchless length routine [4]. We first obtain the number of leading zero bits via `std::countl_zero(x)`, which serves as an index into two precomputed static tables of size 65. The table `digits` stores the candidate decimal length corresponding to each possible leading-zero count, while another table stores the largest integer that still has one fewer digit than that candidate length. A single comparison `x > low` then decides whether the candidate must be incremented, yielding the final digit count. Because the tables are fully static and the only operations are a leading-zero count, two array lookups, and a comparison, the whole function executes in a handful of cycles.

For numbers below 10^{16}, the conversion fits within two 8-digit blocks and is handled entirely by the IFMA kernel (lines 5–10). For larger values, the algorithm divides the input by 10^4, converts the high 16 digits using the same SIMD path, and writes the final four digits with a small scalar helper (`write_four_digits_10000`, lines 13–19). The mask-store operation (line 8 and 17) writes exactly n bytes without conditional branches, ensuring consistent throughput even when digit lengths vary. The same control structure is reused for smaller inputs ($< 10^8$) using an 8-digit variant of our SIMD kernel, analogous to the 16-digit one and omitted here for brevity. This implementation serves as the baseline (*heterogeneous*) variant, balancing performance across heterogeneous digit-length distributions.

The scalar helper `write_four_digits_10000` prints the final four digits by splitting them into two two-digit blocks. Like the table-based methods described in Section 2, it performs a branch-free division by 100 followed by lookups in a precomputed two-digit table. The standard approach to this division uses a 64-bit fixed-point multiplier to compute the quotient $q = (x \cdot \lceil 2^{64}/100 \rceil)$ div 2^{64} and the remainder $r = x - 100 \times q$. While this yields the exact pair (q, r), it requires a relatively costly 64×64→128-bit multiplication. Instead, we use an approximate method that avoids this cost: rather than computing the true remainder, we derive a *pseudo-remainder* using a 32-bit reciprocal and bit shifts. This 8-bit pseudo-remainder takes one of 256 distinct possible values, but each value uniquely corresponds to one of the true remainders in $[0, 100)$. By indexing into a 256-entry lookup table, we recover the correct two-digit

```
int avx512_to_chars(uint64_t value, char *const result) {
    const uint32_t n = fast_digit_count(value);

    // --- Main vectorized path for values below 10^16 ---
    if (value < 10000000000000000ULL) {
        const __m128i digits_15_0 = to_string_avx512ifma(value);
        const __mmask16 mask = (__mmask16)(0xFFFFu << (16 - n));
        _mm_mask_storeu_epi8(result - 16 + n, mask, digits_15_0);
        return n;
    }

    // --- Path for larger values: split into 16+4 digits ---
    const auto [q, r] = div10000(value); // q = value / 10^4, r = value % 10^4
    const uint32_t nq = n - 4; // number of digits in q (r has exactly 4 digits)
    const __m128i v16 = to_string_avx512ifma(q); // as illustrated in Fig. 3
    const __mmask16 mask = (__mmask16)(0xFFFFu << (16 - nq));
    _mm_mask_storeu_epi8(result - 16 + nq, mask, v16);
    write_four_digits_10000(result + nq, r);
    return n;
}
```

FIGURE 6 Simplified heterogeneous conversion path using the IFMA kernel. The full implementation also includes a subpath optimized for when the number has maximum 8-digits that follows the same control logic.

ASCII pair without requiring wide multiplications or subtractions.

5.5 | Homogeneous Specialization

In practical workloads, integer-to-string conversion is rarely applied to a single number. Instead, it is typically performed over a large batch of values. The distribution of digit lengths within such batches can vary widely depending on the application domain. For instance, in some cases, most numbers may have the same number of digits (e.g., all eight-digit values), whereas in others the numbers may span a wide range of lengths.

The previously described heterogeneous variant (§ 5.4) is designed to handle arbitrary digit-length distributions efficiently. However, when the input batch is predominantly composed of numbers with the same digit length, the conversion process can be further optimized by specializing the routine. Instead of using masked stores to handle variable-length outputs, we can emit fixed-size stores.

Masked and unmasked vector stores have identical *throughput* on modern x86 CPUs (e.g., both sustain one store per cycle on the Zen 4 microarchitecture), and although masked stores have higher latency, this difference is hidden when converting numbers in batch.[14] However, a routine that handles variable-length outputs via masked stores must also compute the mask at runtime, calculate the destination pointer offset, and cannot benefit from fixed-size optimizations. These additional instructions add measurable overhead compared to a routine that can assume a fixed

[14] See `https://uops.info/html-instr/VMOVDQU_M128_XMM.html` and `https://uops.info/html-instr/VMOVDQU8_M128_K_XMM.html` for details.

output length and use direct (unmasked) stores.

Our *homogeneous* variant leverages this observation by assuming most numbers in the batch share the same digit length. It contains more branches, each optimized for a specific length. When the dataset is homogeneous, these branches are consistently well predicted and thus nearly free. Once a path is chosen, the body is straight-line code using direct (unmasked) stores with precomputed offsets. Our measurements show that the variable-length (heterogeneous) variant executes approximately 10–12% more instructions per digit, which translates into 30–35% more cycles due to instruction-level dependencies. This design suits common cases such as fixed-width identifiers, Unix timestamps, or telemetry with uniform magnitude.

```
1  int avx512_to_chars_homogeneous_17_20(uint64_t value, char *const result) {
2    // Split value: q = high digits, r = low 16 digits
3    const auto [q, r] = digits::div10e16(value);
4
5    // Write high part (1 to 4 digits) scalarly
6    char *p = digits::write_one_two_three_or_four_digits_10000(result, q);
7
8    // Convert remaining 16 digits using SIMD kernel and store directly
9    const __m128i digits_15_0 = to_string_avx512ifma(r);
10   _mm_storeu_si128(reinterpret_cast<__m128i*>(p), digits_15_0);
11   return p - result + 16; // Total characters = (digits of q) + 16
12 }
```

FIGURE 7 Homogeneous specialization for 17–20-digit inputs. A dispatcher selects this path; within it, the 1–4-digit prefix branches resolve to a single arm in a homogeneous scenario, yielding excellent branch prediction.

Figure 7 shows the fixed path for 17–20-digit numbers. A short dispatcher (not shown) branches on the total digit count to enter this path. Inside the path, the only data-dependent control flow is the 1–4-digit microcase in `write_one_two_three_or_four_digits_10000`. For a fixed total length (17, 18, 19, or 20), the prefix `q` always has 1, 2, 3, or 4 digits, respectively, so the same arm is taken almost every time in a homogeneous scenario, yielding excellent branch prediction. The 16-digit remainder is emitted via a single call to our SIMD algorithm.

5.6 | Dynamic Variant Selection

We implement two specialized variants of our routine: a *homogeneous* variant (branch-heavy, excellent when most numbers share a fixed digit length) and a *heterogeneous* variant (branch-light, slightly more instruction-heavy). The optimal choice depends on the input distribution: logs, counters, or identifiers often cluster around a few magnitudes (e.g., mostly seven- or eight-digit values), whereas numeric data extracted from files or telemetry typically spans a broad range. To adapt automatically, we use a lightweight *dynamic selection* step. Before converting the full batch, we sample a small fraction α of elements and estimate the histogram of digit lengths using the same fast integer-length routine employed by our SIMD algorithm. Let $\rho_{\max}$ denote the top-1 (dominant) length ratio in the sample. If $\rho_{\max}$ exceeds a threshold τ_{hom}, we select the homogeneous variant (with the corresponding fixed length); otherwise we select the heterogeneous variant. Algorithm 1 details the procedure.

The sampling rate α and threshold τ_{hom} are configurable parameters. In our experiments, we use $\alpha = 0.01$ (1%) and

Algorithm 1 Dynamic selection between homogeneous and heterogeneous variants

Require: Iterable data; sampling rate $\alpha \in (0, 1]$; homogeneity threshold $\tau_{\text{hom}} \in (0, 1)$

$m \leftarrow \lceil \alpha \cdot \text{length}(\text{data}) \rceil$
$S \leftarrow \text{RandomSample}(\text{data}, m)$
Initialize $\text{counts}[\ell] \leftarrow 0$ for $\ell = 0 \ldots 20$
for $x \in S$ **do**
    $\ell \leftarrow \text{digit\_length}(x)$ ▷ same routine as in SIMD algorithm
    $\text{counts}[\ell] \leftarrow \text{counts}[\ell] + 1$
$c_{\max} \leftarrow \max_\ell \text{counts}[\ell]$
$\rho_{\max} \leftarrow c_{\max}/m$
if $\rho_{\max} \geq \tau_{\text{hom}}$ **then**
    return Homogeneous
else
    return Heterogeneous

$\tau_{\text{hom}} = 0.95$. The procedure runs in $\Theta(\alpha N)$ time and $\Theta(1)$ extra memory (the histogram is bounded by the maximum digit length, e.g., ≤ 20 bins for 64-bit input). Because $\alpha \ll 1$, the overhead is negligible in practice; quantitative overheads are reported in our evaluation.

6 | EXPERIMENTS

We evaluate our proposed integer-to-string conversion algorithm on two main aspects: (1) the benefits of dynamic variant selection (§ 5.6), and (2) overall performance compared to the most relevant existing methods. We begin by describing our experimental setup, including hardware, datasets, and compared algorithms. We then present our results and analysis.

6.1 | Systems

We conduct our experiments on a machine equipped with an AMD Ryzen 9900X CPU (Zen 5 microarchitecture). The CPU runs at a base frequency of 4.4 GHz and the machine has DDR5 (6000 MT/s) memory. We compile our code using `clang++` 21.1.6 with `-O3 -march=native` optimizations enabled.

We also ran all benchmarks with GCC (`g++` 15.2.1); while the relative ranking of algorithms remains similar, absolute performance differs due to differences in code generation between compilers. The complete `g++` results are provided online in the supplementary materials.

6.2 | Data

We use both real-world datasets and synthetic numbers to evaluate the algorithms. For the real-world datasets, We extract integer data from files commonly used for benchmarking. These files are:

- *Twitter*: an export of data from Twitter's API, 2108 numbers;
- *CITM catalog*: a catalog of events that occurred in a venue part of CitM (Cité de la Musique) in Paris, 14 392 numbers;

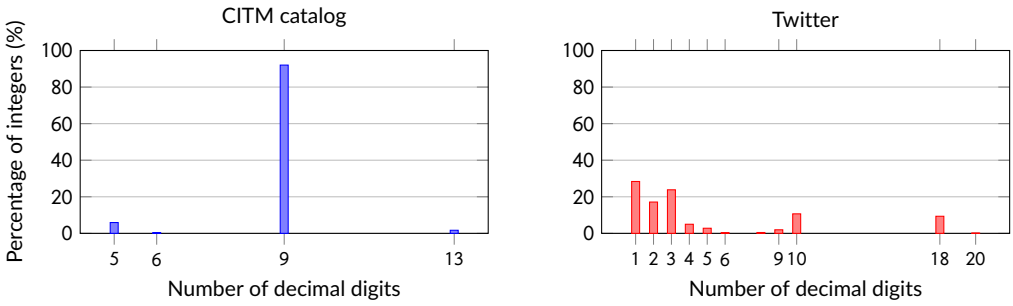

FIGURE 8 Distribution of integer lengths, normalized to percentages. Left: CITM catalog. Right: Twitter.

- *StackOverflow*: a dump of Unix timestamps from StackOverflow posts, 17 823 525 numbers;
- *US patents*: patent IDs from a US patent citation network; represents patents granted between 1975–1999, 16 518 948 numbers.

Figure 8 shows the distribution of integer lengths in the first two datasets. We can see that the CITM catalog is more homogeneous, with 92% of the integers having 9 decimal digits. In contrast, the Twitter data exhibits a more heterogeneous distribution, with significant proportions of small integers (principaly 1–3 digits) as well as larger integers (10 and 18 digits). The other two datasets (StackOverflow timestamps and US patents) are purely homogeneous, with respectively 10-digit and 7-digit integers only.

Synthetic datasets are generated to evaluate algorithmic behavior under controlled digit-length distributions. We consider three synthetic scenarios:

- *uniform*: integers whose decimal digit lengths are sampled uniformly from 1 to 20;
- *natural-8*: integers whose digit lengths are distributed over $[1, 8]$ and strongly concentrated at 8 digits, with a strictly decreasing tail toward shorter representations;
- *natural-16*: the same distributional scheme over $[1, 16]$.

The two *natural* datasets are strongly concentrated at the upper bound. In practice, approximately 96.8% of the generated integers have the maximum digit length. These datasets are designed to mimic the digit-length distribution obtained when sampling integers uniformly from a large interval $[1, 10^k)$: most sampled values have close to the maximum number of digits, while shorter representations occur with exponentially decreasing frequency.

6.3 | Software Implementations

We benchmark a selection of C and C++ libraries converting 64-bit integers to their decimal string representation. Our benchmarking code, synthetic data generators, and datasets are all publicly available online.[15] The benchmarked libraries and algorithms are the following:

- *Naive one-pass*: A baseline implementation based on Figure 1, augmented with the one-pass optimization where the number of digits is precomputed.
- *Champagne–Lemire*: Our algorithm described in Section 5. *Unless otherwise specified, we use the version employing dynamic variant selection (which automatically chooses between the homogeneous and heterogeneous variants).*

[15] `https://github.com/fastfloat/int_serialization_benchmark`

- *Abseil*: Google's Abseil implementation via the `FastIntToBuffer` function.[16]
- *jeaiii*: The `to_text_from_integer` function from `jeaiii`'s `itoa` library.[17]
- *yy_itoa*: The `itoa_u64_yy` function by Yaoyuan Guo.[18]
- *AppNexus*: The `itoa_u64_an` function from the AppNexus Common Framework library.[19]
- *Hopman*: An implementation of the `hopman_fast` algorithm.[20] We reimplemented the algorithm to support 64-bit integers (the original only supports 32-bit inputs) and to match the interface used by our other implementations (accepting a buffer pointer and returning the output length).
- *Mathisen*: An SSE4.1-based algorithm inspired by Mathisen's approach, based on the "four groups of three digits" variant described by the StackOverflow user `icecreamsword`. Our implementation decomposes the input into base-10^9 chunks, prints the most significant chunk without leading zeros, and formats the remaining chunks as fixed-width blocks using SIMD digit extraction. We could not include Mathisen's original code, as it does not appear to be publicly available and only supports 32-bit integers.
- `std::to_chars`: The C++17 standard library integer-to-string conversion function.

All of these algorithms are described in more details in Section 2.

We exclude Muła's SSE-based `utoa64_sse` algorithm [23][21] from our benchmarks because its interface differs from the others: it may write digits at arbitrary positions within the output buffer rather than starting at the beginning. Adapting it to match our interface would require either copying the result to the buffer start (underestimating performance) or using AVX-512 masked stores (making it no longer a pure SSE algorithm).

6.4 | Results – Dynamic Variant Selection

We first assess whether the dual-variant strategy is worthwhile by measuring (1) the overhead of the selection mechanism itself and (2) the performance gains it unlocks. We focus on two representative synthetic distributions: *uniform* (highly heterogeneous, 1–20 digits) and *natural-16* (strongly homogeneous, 96.8% at 16 digits). These extremes bracket the spectrum of real-world digit-length distributions.

6.4.1 | Dynamic Selection Overhead

Table 1 reports the absolute time spent in Algorithm 1 (sampling 1% of the input) compared to the total conversion time for datasets of varying sizes. The rightmost column shows the ratio between conversion time and selection overhead.

The overhead ratios range from approximately 1 500 (natural-16) to 12 500 (uniform), reflecting the fact that heterogeneous data incurs higher conversion cost (due to branch mispredictions and variable-length handling) while selection cost remains similar. In the worst case—homogeneous data, where selection represents the largest *relative* overhead—it consumes only 0.067% of total runtime. Because this ratio is nearly constant across dataset sizes, the selection step scales linearly with negligible marginal cost. Consequently, always enabling auto-selection incurs no significant penalty in practice, even when the choice between variants ultimately has little impact.

[16] `https://github.com/abseil/abseil-cpp.git`, git hash `d7aaad8` (April 2024).

[17] `https://github.com/jeaiii/itoa/blob/c861d1c/include/itoa/jeaiii_to_text.h`, git hash `c861d1c` (November 2022).

[18] `https://github.com/ibireme/c_numconv_benchmark/blob/8541662/src/itoa/itoa_yy.c`, git hash `8541662` (August 2024).

[19] `https://github.com/appnexus/acf/blob/5a4645d/src/an_itoa.c`, git hash `5a4645d` (December 2017).

[20] `https://stackoverflow.com/a/4364057`

[21] `https://github.com/WojciechMula/toys/blob/7731566/sse-utoa/sse64-intrin.c`

TABLE 1 Evaluating the variant-selection overhead compared to the entire integer-to-string conversion of homogeneous and heterogeneous datasets (g++).

Model	Size	Variant Selection Time (ns)	Integer-to-string Time (ns)	Ratio
uniform	100 000	38	479 000	12 605
	1 000 000	390	4 950 000	12 692
	10 000 000	3 962	49 700 000	12 544
natural-16	100 000	110	169 000	1 536
	1 000 000	1 137	1 730 000	1 521
	10 000 000	11 922	17 500 000	1 468

TABLE 2 Performance comparison of homogeneous vs heterogeneous variants across different datasets. Times shown in nanoseconds per number (ns/n). The **bold** value indicates the faster variant for each dataset. The "Selected" column shows which variant was chosen by auto-detection.

Dataset	Homogeneous (ns/n)	Heterogeneous (ns/n)	Selected
Uniform 4-digit (1M)	**0.73**	0.75	Homo
Uniform 8-digit (1M)	0.9	**0.72**	Homo
Uniform 16-digit (1M)	**0.93**	1.19	Homo
Uniform 1-20 digits (1M)	7.86	**4.56**	Hetero
Natural-8 (1M)	1.08	**0.73**	Homo
Natural-16 (1M)	**1.07**	1.2	Homo
CITM Catalog	1.4	**1.23**	Homo
Twitter JSON	1.03	**0.93**	Hetero
StackOverflow Timestamps	1.28	**1.26**	Homo
US Patents	**0.95**	0.96	Homo

6.4.2 | Performance Impact of Variant Choice

Having established that selection overhead is negligible, we quantify the performance benefit of choosing the right variant. Table 2 compares the per-integer conversion time (ns/n) of the heterogeneous (§ 5.4) and homogeneous (§ 5.5) variants across both synthetic and real-world datasets. The "Selected" column indicates which variant the auto-detection heuristic selects for each dataset.

On uniform 4-digit data, the homogeneous variant outperforms the heterogeneous one (0.73 vs. 0.75 ns/n, 3%). The gap widens for 16-digit data (0.93 vs. 1.19 ns/n, 28%), consistent with the reduced instruction count described in § 5.5. The auto-detection heuristic correctly selects the homogeneous variant in both cases. Curiously, the 8-digit case reverses this trend under Clang: the heterogeneous variant is faster (0.72 vs. 0.90 ns/n). Performance counters reveal a compiler code-generation effect: Clang's heterogeneous code has fewer branches (2 vs. 5) and more instructions per cycle (6.6 vs. 5.4), overcoming the single extra instruction. The heuristic misclassifies this case, incurring a 25%

penalty—and the predominantly 8-digit Natural-8 dataset suffers similarly (48%). This anomaly is compiler-specific: under GCC, the homogeneous variant wins (0.75 vs. 0.96 ns/n). More broadly, GCC produces a slower heterogeneous variant across most datasets (e.g., 1.03 vs. 0.75 ns/n on 4-digit), making the homogeneous variant relatively more attractive.

Conversely, the highly mixed uniform 1–20 digit distribution strongly favors the heterogeneous variant: 4.56 ns/n vs. 7.86 ns/n (72% speedup). Branch mispredictions in the homogeneous dispatcher outweigh any store optimization, confirming that branch-light code is essential when digit lengths vary unpredictably. The heuristic correctly identifies this case.

Real-world datasets show smaller differences: CITM catalog, Twitter, StackOverflow, and US patents all exhibit gaps under 14%. In these cases, misclassification penalties remain modest (8–10%).

These results validate the dual-variant design: substantial speedups (up to 72%) when input characteristics clearly favor one implementation, with minimal cost when both variants perform similarly. The adaptive strategy ensures robust performance across diverse workloads without manual tuning.

6.5 | Results – Performance Comparison of Tested Algorithms

We now compare the overall performance of our proposed algorithm (with dynamic variant selection enabled) against the other tested algorithms.

6.5.1 | Heterogeneous and Real-World Datasets

Figure 9 compares the performance of the most competitive algorithms across three datasets: Twitter JSON (variable-length, 69% with $\leq$3 digits), US patents (homogeneous 7-digit identifiers), and Natural 1-8 (uniform distribution over 1–8 digits). Our AVX-512 approach consistently outperforms all competitors. On the Natural 1-8 dataset, we achieve 0.73 ns/n, corresponding to a throughput of 1.4 GB/s of string output—nearly twice the rate of the next best competitor (`jeaiii` at 0.75 GB/s). The performance gap is particularly pronounced on homogeneous datasets: on US patents, `jeaiii` is 39% slower, and on Natural 1-8, it is 81% slower. Even on Twitter—where scalar approaches traditionally excel due to the prevalence of small integers—our algorithm maintains a 33% advantage.

6.5.2 | Homogeneous Fixed Digit-Length Datasets

Fixed digit-length benchmarks are common, though they should be interpreted with care: an algorithm that performs well on each fixed length individually may still underperform on variable-length workloads due to branch mispredictions or mode-switching overhead. Nevertheless, homogeneous datasets do occur in practice—database identifiers, timestamps, and telemetry counters often cluster around a single magnitude—and our previous experiments on real-world datasets already demonstrate strong performance under variable-length conditions. We therefore include fixed-length benchmarks to provide a complete picture of per-length behavior.

Figure 10 reports results across all digit lengths (1 to 20 digits). For single-digit numbers, `std::to_chars` is fastest at 0.42 ns/n, compared to 1.25 ns/n for our algorithm. For 2-digit numbers, `jeaiii` leads at 0.72 ns/n versus 1.08 ns/n for ours. At 3 digits we are essentially tied with `jeaiii` (0.90 vs. 0.91 ns/n), but starting at 4 digits we pull decisively ahead and dominate all competitors across all remaining sizes. The advantage is particularly striking at 8 digits, where our algorithm achieves 0.72 ns/n—an 85% speedup over `jeaiii` (1.33 ns/n). Beyond 10 digits, `jeaiii`'s performance becomes erratic, with notable slowdowns at 10 and 18 digits. These results demonstrate that

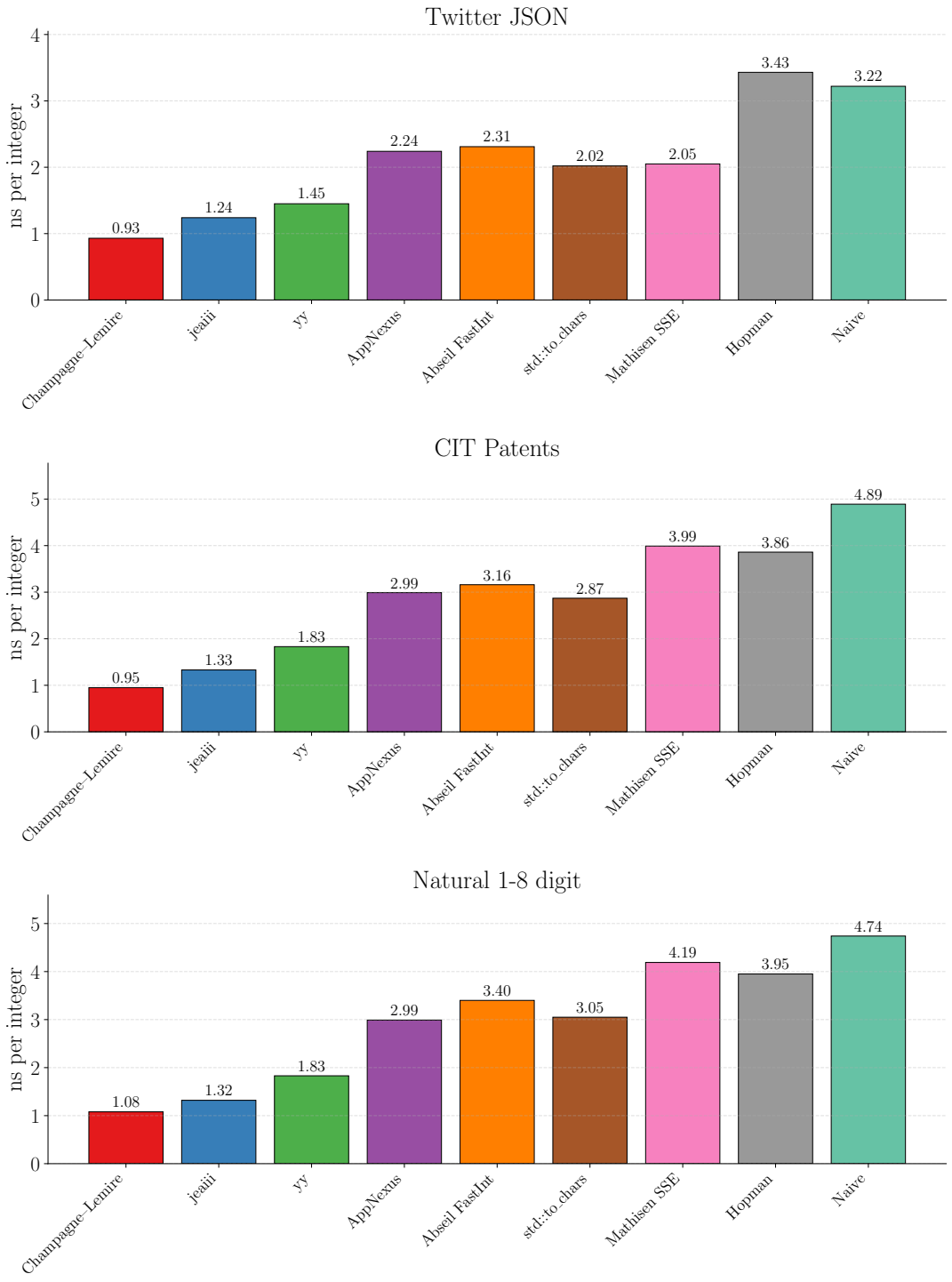

FIGURE 9 Performance comparison (ns/n, lower is better) of top algorithms on three real-world datasets. Our AVX-512 approach outperforms all competitors, achieving 1.4 GB/s throughput on Natural 1-8. Extended results with instruction and cycle counts are provided in the supplementary materials.

our algorithm's performance advantage holds across the full spectrum of digit lengths, not just on specific real-world distributions.

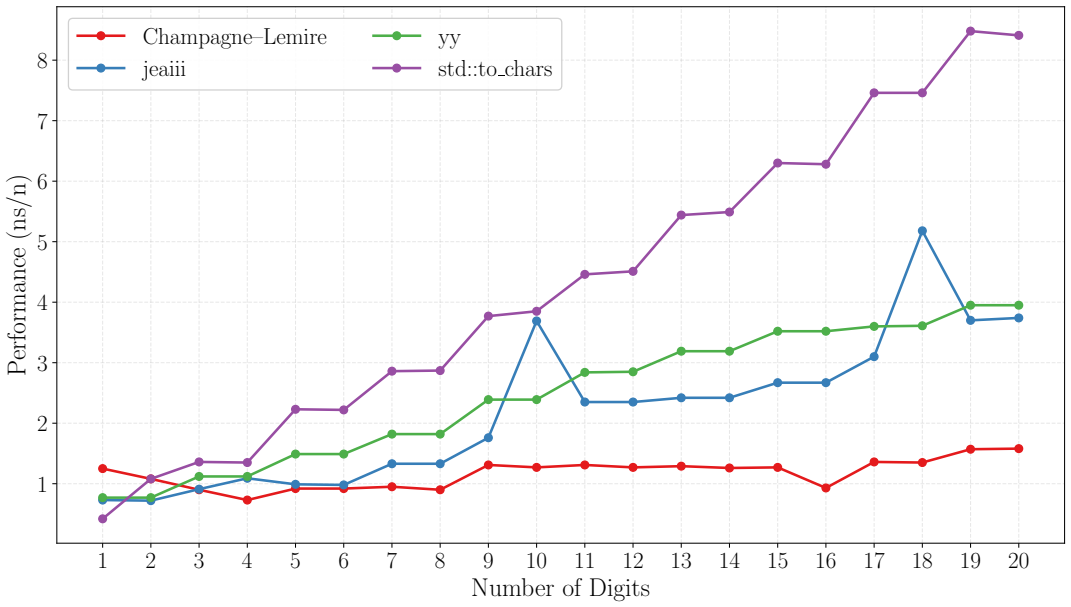

FIGURE 10 Performance across all digit lengths (1–20) on homogeneous datasets of 1M integers each. Our AVX-512 algorithm dominates from 3 digits onward. A complete comparison including all algorithms is available in the supplementary materials.

7 | CONCLUSION

We presented the first AVX-512 integer-to-string conversion algorithm in the literature. Our approach leverages AVX-512 IFMA instructions to extract multiple digits in parallel, achieving high throughput across nearly all digit lengths. To handle variable-length outputs efficiently, we designed both heterogeneous (branch-light, masked stores) and homogeneous (branch-heavy, unmasked stores) variants, along with a dynamic selection mechanism that adaptively chooses the optimal implementation based on input characteristics. Our experiments demonstrate that this dual-variant strategy delivers substantial speedups (up to 72%) on synthetic datasets with extreme digit-length distributions, while imposing negligible overhead (0.067% worst-case) on real-world data where both variants perform similarly. Overall, our AVX-512 algorithm outperforms all existing state-of-the-art methods across all tested datasets, running up to twice as fast as the best competing scalar approach (`jeaiii`) and up to four times faster than the standard library's `std::to_chars`. This work highlights the potential of SIMD techniques for accelerating integer-to-string conversion and provides a robust, high-performance solution across diverse workloads.

Two natural extensions of this work deserve further investigation. First, while our implementation targets AVX-512 IFMA instructions, it would be valuable to evaluate how the proposed SIMD-based approach maps to alternative vector architectures, in particular ARM platforms supporting SVE or SVE2. Although the available instruction sets

differ, the underlying parallel digit extraction strategy may transfer with suitable adaptations. Second, our study focuses on converting one integer at a time; exploring SIMD-oriented approaches that convert groups of integers in batch could enable a different class of vectorization strategies, potentially improving throughput in data-parallel serialization workloads.

Although our study focuses on integer-to-string conversion, the problem is closely related to floating-point number formatting, which is inherently more complex. In floating-point conversion, one must first determine the correct decimal exponent and mantissa from the binary representation before producing the textual output. The final stage of this process—writing the decimal mantissa—is conceptually similar to integer-to-string conversion, but over a smaller range (e.g., up to seventeen digits for IEEE 754 double-precision values versus up to twenty digits for 64-bit unsigned integers). Additional challenges include placing the decimal point, supporting both fixed and scientific notations, and ensuring rounding correctness. While these aspects require extra handling, the core operation—converting the decimal mantissa to its textual form—shares the same computational bottlenecks as integer-to-string conversion. Consequently, advances in integer conversion techniques can accelerate part of the core of floating-point formatting routines, with adaptations needed to address their specific requirements.

Author Contributions

Jaël Champagne Gareau: conceptualization; investigation; software; experimentation; writing-review and editing. Daniel Lemire: conceptualization; software; validation; experimentation; data analysis; writing-original draft; writing-review and editing.

Funding Information

This work was supported by the Natural Sciences and Engineering Research Council of Canada, Grant Number: RGPIN-2024-03787. The first author is supported by a postdoctoral grant from Fonds de recherche du Québec, `https://doi.org/10.69777/361128`.

Data Availability Statement

All our data and software is freely available online. The C++ benchmarking software, along with the datasets used, is available at `https://github.com/fastfloat/int_serialization_benchmark`. The supplementary materials referenced throughout this paper, including complete performance tables and raw benchmark data, are available on the paper's webpage at `https://www.jaelgareau.com/en/publication/gareau_lemire-spe26/`.

References

[1] Clausecker R, Lemire D. Transcoding unicode characters with AVX-512 instructions. Software: Practice and Experience 2023;53(12):2430–2462.

[2] Muła W, Lemire D. Base64 encoding and decoding at almost the speed of a memory copy. Software: Practice and Experience 2020;50(2):89–97.

[3] Kernighan BW, Ritchie DM. The C Programming Language. 2nd ed. Englewood Cliffs, NJ: Prentice-Hall; 1988.

[4] Lemire D, Counting the digits of 64-bit integers; 2025. `https://lemire.me/blog/2025/01/07/counting-the-digits-of-64-bit-integers/`, accessed: 14 January 2026.

[5] Alexandrescu A, Three Optimization Tips for C++ Code; 2012. Video recording available at https://archive.org/details/AndreiAlexandrescu-Three-Optimization-Tips. Associated blog post: https://www.facebook.com/notes/10158791579037200/ Accessed: 14 January 2026.

[6] Fog A, Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs; 2025. https://www.agner.org/optimize/instruction_tables.pdf, accessed: 14 January 2026.

[7] Jacobsohn DH. A combinatoric division algorithm for fixed-integer divisors. IEEE Trans on Comput 1973;100(6):608–610.

[8] Artzy E, Hinds JA, Saal HJ. A Fast Division Technique for Constant Divisors. Commun ACM 1976 Feb;19(2):98–101.

[9] Li SYR. Fast Constant Division Routines. IEEE Trans Comput 1985 Sep;34(9):866–869.

[10] Magenheimer DJ, Peters L, Pettis K, Zuras D. Integer Multiplication and Division on the HP Precision Architecture. SIGARCH Comput Archit News 1987 Oct;15(5):90–99. http://doi.acm.org/10.1145/36177.36189.

[11] Robison AD. N-Bit Unsigned Division via N-Bit Multiply-Add. In: Proceedings of the 17th IEEE Symposium on Computer Arithmetic ARITH '05, Washington, DC, USA: IEEE Computer Society; 2005. p. 131–139.

[12] Granlund T, Montgomery PL. Division by Invariant Integers Using Multiplication. SIGPLAN Not 1994 Jun;29(6):61–72.

[13] Cavagnino D, Werbrouck AE. Efficient Algorithms for Integer Division by Constants Using Multiplication. Comput J 2008 Jul;51(4):470–480.

[14] Warren HS Jr. Hacker's Delight. 2nd ed. Boston: Addison-Wesley; 2013.

[15] Lemire D, Bartlett C, Kaser O. Integer division by constants: optimal bounds. Heliyon 2021;7(6).

[16] Chateauneu R, Good old Integer To ASCII conversion (itoa). SourceForge; 2007. https://sourceforge.net/projects/itoa/, GPL-2.0 licensed project for fast integer-to-string conversion, Accessed: 14 January 2026.

[17] Henriksen IE, Answer to "optimized itoa function"; 2013. https://stackoverflow.com/a/15461080/5183410, answer embeds code that attributes the optimization to Inge Eivind Henriksen (2013) and the original itoa to James Clark/groff. Accessed: 14 January 2026.

[18] fmtlib contributors, format-benchmark: u2985907 implementation; 2020. https://github.com/fmtlib/format-benchmark/blob/master/src/u2985907.h, accessed: 14 January 2026. GitHub repository.

[19] Hopman C, Answer to "C++ performance challenge: integer to std::string conversion"; 2010. https://stackoverflow.com/a/4364057, StackOverflow answer describing the 'hopman_fun' (arithmetic) and 'hopman_fast' (10^4 lookup) variants.

[20] Mathisen T, Fast integer to ASCII conversion code; 1999. https://groups.google.com/g/comp.lang.asm.x86/c/rBzgsQdkq8c/m/JCv8EkPgCS4J, usenet post, Accessed: 14 January 2026. Message posted to comp.lang.asm.x86 newsgroup.

[21] Mathisen T, Converting numbers to text; 2015. https://comp.lang.asm.x86.narkive.com/hIuZMYCO/converting-numbers-to-text#post2, usenet post, Accessed: 14 January 2026. Message posted to comp.lang.asm.x86 newsgroup.

[22] Khuong P, How to print integers really fast (with Open Source AppNexus code!); 2017. https://pvk.ca/Blog/2017/12/22/appnexus-common-framework-its-out-also-how-to-print-integers-faster/, accessed: 14 January 2026.

[23] Muła W, SSE: conversion integers to decimal representation; 2011. http://0x80.pl/notesen/2011-10-21-sse-itoa.html, accessed: 14 January 2026.

[24] User *icecreamsword*, Answer to "optimized itoa function"; 2015. https://stackoverflow.com/a/32818030/5183410, StackOverflow answer describing a SIMD variant of Mathisen's algorithm, Accessed: 14 January 2026.

[25] Adams U. Ryū: Fast Float-to-String Conversion. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI 2018, New York, NY, USA: Association for Computing Machinery; 2018. p. 270–282.

[26] Lemire D, Kaser O, Kurz N. Faster remainder by direct computation: Applications to compilers and software libraries. Software: Practice and Experience 2019;49(6):953–970.

A | LIST OF ACRONYMS

Acronym	Full form
ASCII	American Standard Code for Information Interchange
AVX	Advanced Vector Extensions
AVX-512	Advanced Vector Extensions 512-bit
AVX-512BW	AVX-512 Byte and Word
AVX-512CD	AVX-512 Conflict Detection
AVX-512DQ	AVX-512 Doubleword and Quadword
AVX-512F	AVX-512 Foundation
AVX-512IFMA	AVX-512 Integer Fused Multiply-Add
AVX-512VL	AVX-512 Vector Length
AVX2	Advanced Vector Extensions 2
CSV	Comma-Separated Values
EVEX	Enhanced Vector Extensions (AVX-512 prefix)
FMA	Fused Multiply-Add
IFMA	Integer Fused Multiply-Add
ns	nanoseconds
ns/n	nanoseconds per integer
SIMD	Single Instruction, Multiple Data
SSE	Streaming SIMD Extensions
SSE2	Streaming SIMD Extensions 2
SSE4.1	Streaming SIMD Extensions 4.1