Journal of
Low Power Electronics and Applications

MDPI

Article

Energy-Efficient SIMD-Accelerated Expression Parsing: A Cross-Architectural Microarchitectural Analysis in Managed Environments

Pavel Kyurkchiev [1,*] and Anton Iliev [1,2]

[1] Faculty of Mathematics and Informatics, University of Plovdiv "Paisii Hilendarski", 24, Tzar Asen Str., 4000 Plovdiv, Bulgaria; aii@uni-plovdiv.bg
[2] Centre of Excellence in Informatics and Information and Communication Technologies, 1113 Sofia, Bulgaria
* Correspondence: pkyurkchiev@uni-plovdiv.bg

Abstract

Mathematical expression evaluation is a critical component in high-performance software systems. Traditional state-dependent algorithms resist Single Instruction, Multiple Data vectorization, while managed runtime environments introduce Garbage Collection overhead. This paper presents a hybrid parsing pipeline that isolates sequential state operations from parallelizable tasks, combining a SIMD-accelerated lexical analyzer with a vector evaluator that allocates no managed memory in its inner loop. Against established .NET evaluators the pipeline leads only below a few hundred evaluations of a given expression (approximately 230 on AMD and 310 on Intel), above which compiling once to a delegate dominates, and vectorized tokenization inverts beyond roughly thirty thousand characters of input. AVX-2 (Vector256) instructions reduce the isolated evaluation of a seven-operand reference expression to 69.30 ns with zero bytes allocated, while the complete pipeline allocates 1216 bytes and remains subject to generational collection. Under a fixed duty cycle holding delivered work constant, active-state residency falls from 69.1% to 51.6% and package energy falls by 15.9%, a measured effect rather than an inferred one. Benchmark-synchronized power instrumentation shows that the energy cost of wider vectors on AMD Zen 4 is expressed in time rather than in power. On both platforms 512-bit packed operations essentially retain the latency of their 256-bit counterparts but consume twice the issue slots. In the evaluation workload they take longer. Where lanes are fully occupied and operations independent, the ordering reverses and the 512-bit path draws less power and consumes 23.8% less energy per element. Power and energy were measured on the AMD platform only. The Intel platform supports the latency and allocation comparison.

Keywords: SIMD vectorization; mathematical expression evaluation; managed runtime environments; Energy-Delay Product (EDP); microarchitectural analysis; Garbage Collection overhead; AVX-2 and AVX-512

Check for updates

Academic Editors: Spyridon Nikolaidis and Ronald Tetzlaff

Received: 14 August 2026
Revised: 5 October 2026
Accepted: 8 October 2026
Published: 9 October 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

Mathematical expression evaluation is a critical component in various high performance software systems. As data processing demands escalate in modern computing, achieving both low latency and high energy efficiency has become paramount. However, traditional evaluation pipelines rely heavily on state-dependent algorithms, such as character-by-character lexical analysis and the stack-based Shunting Yard algorithm, which inherently resist Single Instruction, Multiple Data (SIMD) vectorization. Fundamental

research into memory management paradigms confirms that while automated collection provides significant software engineering benefits, it introduces execution overheads that vary substantially with heap size and collector choice [1].

In managed runtime environments (MREs), these algorithmic bottlenecks are further exacerbated by Garbage Collection (GC) overhead. The continuous allocation of intermediate artifacts, such as token streams and syntax trees, triggers frequent GC cycles. As demonstrated in prior architectural studies on managed software, the power and performance profile of runtime services such as collection differs markedly from that of application code, so the two respond differently to the underlying hardware [2]. Moreover, evaluating such runtime behaviour credibly requires measurement on real silicon rather than architectural models, since architectural simulators diverge systematically from the hardware they model, and the divergence is rarely quantified against it [3].

To overcome these inherent limitations, we propose a hybrid pipeline architecture specifically designed to decouple branch-heavy sequential state transitions from arithmetic that can be mapped onto vector registers. Integrating a custom SIMD-based lexical analyzer with a vector evaluator that allocates no managed memory in its inner loop, our approach aims to actively suppress dynamic power dissipation. We establish a robust experimental baseline using the BenchmarkDotNet framework to isolate and quantify the latency of each discrete parsing stage (tokenization, conversion, and evaluation) against standard scalar implementations. Such stage-level isolation is necessary because the execution characteristics of managed runtimes diverge substantially from those of natively compiled code, a divergence documented since the earliest architectural studies of virtual machine execution [4].

A central contribution of this research is the execution of a cross-architectural evaluation across modern AMD Zen and Intel Core mobile platforms. Hardware profiling shows that targeting the AVX-2 (Vector256) instruction set reduces the isolated evaluation of the reference expression to 69.30 ns with no managed allocation, as measured under BenchmarkDotNet (Section 2.1). Shorter execution shortens the interval during which the core must remain in the active C0 state, and we test this directly rather than by inference. Under a fixed duty cycle delivering identical work, the proposed pipeline reduces active residency from 69.1% to 51.6%, raising idle residency by 17.5 percentage points.

Additionally, utilizing the Energy-Delay Product (EDP) as a proxy metric [5], this study investigates hardware-specific execution anomalies associated with the RyuJIT compiler. We expose counterintuitive latency regressions when scaling to 512-bit vector paths, and isolate their mechanism with a dedicated instruction-level experiment measuring latency and reciprocal throughput at 128, 256 and 512 bits with a distinct value in every lane. On both platforms this experiment shows that 512-bit operations retain the latency of their 256-bit counterparts but consume twice the issue slots, through double-pumping on AMD and through a single port-fused 512-bit unit on the Intel client core. Benchmark-synchronized power measurement then separates a consequence that latency figures alone conflate. On AMD, where lanes are fully occupied, the halved issue rate that lengthens execution also lowers switching activity, so the wider path draws less power and its energy cost is markedly smaller than its time cost. The empirical findings underscore that maximizing efficiency in managed runtimes requires distinguishing latency from energy and matching vector width to lane occupancy and dependency structure, rather than defaulting to the widest available register.

2. Materials and Methods

2.1. Experimental Setup and Benchmarking Framework

Evaluating microarchitectural performance in MREs requires rigorous methodologies to isolate algorithmic execution from non-deterministic background processes, such as Just-In-Time (JIT) compilation, GC heuristics, and Operating System (OS) context switching. To achieve statistically significant and reproducible results, the experimental evaluation was conducted using BenchmarkDotNet (v0.15.8), an industry-standard, high-precision micro-benchmarking framework for the .NET ecosystem.

The benchmark harness is configured via a custom `ManualConfig` instance that composes the default BenchmarkDotNet toolchain (including column providers, exporters, analysers, jobs, validators, filters, and logical grouping rules). The evaluation employs three instruments, each measuring a quantity the others cannot, under conditions appropriate to that quantity.

1. **Latency and allocation.** A single BenchmarkDotNet session executes six fixtures through `BenchmarkSwitcher` (baseline, amortisation, depth sweep, length sweep, error corpus and vector-width microbenchmarks), comprising approximately 161 cases. The fixtures remain distinct types so that their parameter sets do not form a Cartesian product, but the session is a single invocation writing to one timestamped artefact directory, so that a repeated run cannot overwrite the data behind an already published table. Each case is measured 30 times, as 10 iterations across 3 process launches at a fixed warmup of 3. Three launches are used rather than one long run because only separate processes capture between-process variation such as JIT decisions, memory layout and address-space randomisation. A single process with many iterations measures within-process noise and understates the confidence interval. Allocation and generational collection counts are captured by `[MemoryDiagnoser]` through Event Tracing for Windows (ETW).
2. **Power and frequency.** Power cannot be measured on this suite. BenchmarkDotNet executes short bursts, whereas AMD uProf samples at 10 to 100 ms and requires a variant to occupy the core for tens of seconds. A separate harness therefore holds each variant in sustained execution, delegating to the live benchmark instance after `[GlobalSetup]` rather than reimplementing it. Eight segments (scalar, 256-bit and 512-bit evaluation, three full pipelines, two throughput kernels) run for 45 s each, each preceded by 3 s of warmup and followed by 10 s of separation, bracketed by two 15 s idle windows, for a total of 494 s. The session is repeated in three segment orderings (forward, reverse, and shuffled under a fixed seed), separated by 120 s of cooling, because a segment following a heavy AVX-512 run starts on a hotter core. A difference visible in only one ordering is a thermal artefact. The harness emits UTC begin and end markers against which the uProf trace is cut, so that power samples are attributed to variants rather than averaged across the session.
3. **Duty cycle.** A third harness runs two segments, the same delivered work scalar against SIMD, under a fixed 10 ms duty cycle, isolating active-state residency at constant throughput (Section 3.10.2).

Both sustained harnesses pin the process to one logical CPU whose simultaneous multithreading sibling is left idle, raise process priority, and disable tiered compilation, tiered profile-guided optimisation and ReadyToRun, so that the emitted code cannot change shape mid-segment. These settings shift absolute timings substantially, by a factor of roughly three for the scalar evaluation baseline, and latencies from the two instruments are therefore not interchangeable. Validation references were captured in this same environment, and after each run the harness compares the measured ns/op against them, raising

a warning beyond ±10%. A segment that deviates is discarded, since its power figures would describe different code.

To stabilize CPU frequency and minimize dynamic voltage and frequency scaling (DVFS) artifacts, the framework explicitly forced the OS power plan to "High Performance" prior to execution. While this approach guarantees measurement stability, we acknowledge that in a real-world deployment, aggressive DVFS scaling and the inherent latencies of transitioning into and out of deep C-states could alter the exact absolute energy footprints calculated under this profile.

The execution pipeline for each BenchmarkDotNet run was segmented into discrete phases to guarantee measurements were captured only after the execution path was fully optimized by the Tier-1 RyuJIT compiler:

1. **Overhead Evaluation:** Pilot and warmup iterations strictly designed to measure and subsequently subtract the benchmarking infrastructure's own invocation overhead.
2. **Workload Warmup:** Iterations executed to populate the L1/L2 instruction caches, train the branch prediction unit, and force the JIT compiler to emit optimally vectorized machine code.
3. **Workload Actual:** The core measurement phase, executing millions of operations (e.g., up to 8,388,608 operations per iteration for high-throughput Vector512 scenarios) to compute stable statistical distributions.

To ensure statistical robustness, the framework automatically applied heuristic filters to identify and remove environmental outliers. Furthermore, the statistical engine detected multimodal (bimodal) distributions in highly complex end-to-end scenarios (exhibiting M-Values > 3.0). Such bimodality indicates distinct execution paths within the JIT-compiled code or varying GC pause durations, highlighting the necessity of utilizing median values and 99.9% Confidence Intervals (CI) alongside standard arithmetic means to accurately interpret energy-delay characteristics, as strongly recommended in managed runtime performance evaluation methodologies [6]. Failing to account for these environmental variations often leads to measurement bias and incorrect hardware evaluation conclusions [7]. Beyond interval estimation, pairwise comparisons between configurations are assessed with Welch's two-sample t-test on $n = 30$ measurements per configuration (10 iterations across 3 process launches), which does not assume equal variances. Effect sizes are reported as Hedges' g with the small-sample correction applied. Where a distribution is visibly right-skewed, the comparison is additionally reported on medians and the skew is stated explicitly. Given the number of planned comparisons, significance thresholds are interpreted under Holm–Bonferroni correction.

2.2. Diagnostic Tooling and Memory Instrumentation

To analyze the energy-efficiency characteristics required for this study, the default timing metrics were augmented with the `[MemoryDiagnoser]` profiler, which intercepts ETW allocation events to quantify managed heap pressure per operation. Architectural-level power analysis frameworks have long established that clock cycling and active functional units are the primary drivers of dynamic energy dissipation [8]. The specific configurations and extracted metrics are detailed in Table 1.

2.3. Workload Design and Expression Complexity

To eliminate variations caused by differing input lengths and structural complexities, a fixed, representative arithmetic expression was continuously evaluated across all isolated parsing stages and non-batch scenarios:

$$(1 + 2) * 3 - (2.2 * 1.1)/0.5 + 0.2 \quad (1)$$

Table 1. Benchmarking framework configuration and extracted diagnostic metrics.

Category	Metric/Parameter	Description and Relevance to Energy Efficiency
Power Profiling	Package Power (W)	Continuous physical power draw sampled via AMD uProf during the sustained-power sessions. Provides the basis for Energy (E) calculations, which use the dynamic term above a measured idle baseline rather than package draw directly.
	Core Frequency (MHz)	Continuous active core clock speed recorded to observe DVFS interventions and thermal throttling during SIMD execution.
Execution Control	Process Isolation	Dedicated child process per benchmark to eliminate memory fragmentation.
	JIT Optimization	Pre-compiled at Tier-1, prevents background JIT compilation during measurement.
Statistical Metrics	Mean & StdDev	Arithmetic mean and standard deviation of execution time (in nanoseconds/microseconds).
	Confidence Interval	99.9% CI bounding for latency stability evaluation.
Memory Diagnostics	Allocated Memory	Tracked in Bytes per Operation. Reported separately for each pipeline stage, since only the isolated evaluation stage is allocation-free.
	Gen 0/1/2 Collections	Counts GC triggers per 1000 operations. High GC activity correlates with CPU power spikes.

Despite its brevity, this expression was meticulously constructed to evaluate the robustness of the parsing pipeline by encompassing:

- **Full Operator Precedence:** Utilizing addition, subtraction, multiplication, and division to trigger complex state transitions in the Shunting Yard baseline.
- **Nested Grouping:** Parenthetical structures forcing recursive evaluations or stack pushes.
- **Mixed-Type Coercion:** Combining integer tokens (`1`, `2`, `3`) with floating-point tokens (`2.2`, `1.1`, `0.5`, `0.2`) to stress the lexical analyzer's numeric parsing boundaries and SIMD lane alignment.

By pre-computing the initial states for isolated benchmarks (e.g., passing pre-tokenized arrays directly to the `ConvertOnly` and `EvalOnly` categories), the methodology ensures that structural complexities only stress the specifically targeted algorithm, preventing downstream latency masking.

Two properties of this expression bound the interpretation of the vectorized evaluation results and are stated here explicitly. First, each of the six binary operations in the expression yields a single useful result, and the evaluator broadcasts each operand across all lanes (Section 2.5), so a wider register does not increase the useful work per operation. Second, postfix evaluation proceeds through a stack-carried dependency chain, so consecutive arithmetic operations cannot be issued concurrently and the workload is governed by instruction latency rather than by issue throughput.

Because a single expression cannot characterize the pipeline on its own, three additional corpora were generated for the experiments reported in Sections 3.7 and 3.8. A *length sweep* varies input length across 40, 131, 517, 2050, 8196 and 32,774 characters at a fixed nesting depth of four. A *depth sweep* holds length near 258 characters and varies nesting depth across 1, 2, 4, 8 and 15, in both left-associated and right-associated forms, so that structural depth is decoupled from token count. An *error corpus* of twelve malformed inputs exercises the rejection path at varying positions within the input, from position zero to end-of-input. Unary operators and transcendental functions lie outside the supported grammar (Section 2.5) and therefore outside the performance corpora, and operator distribution is held fixed (Section 4.4).

2.4. Hardware and Runtime Environments

To perform a rigorous cross-architectural analysis, the benchmark suite was executed on two distinct modern microarchitectures. The detailed specifications of both hardware platforms, along with the shared software ecosystem, are summarized in Table 2.

Table 2. Experimental hardware and software specifications.

Category	Component/Parameter	Specification
Hardware (AMD)	CPU	AMD Ryzen 7 PRO 7840U (8 physical/16 logical cores, 3.30 GHz, up to 5.10 GHz)
	Cache Hierarchy	16 MB L3/8 MB L2
	Thermal Design Power	15–28 W
	Instruction Sets	AVX, AVX2, AVX-512 (Zen 4 microarchitecture)
	RAM	32 GB DDR5 6400 MT/s
Hardware (Intel)	CPU	Intel Core i5-11400H (6 physical/12 logical cores, up to 4.50 GHz)
	Cache Hierarchy	12 MB L3 (Intel Smart Cache)
	Thermal Design Power	45 W
	Instruction Sets	AVX, AVX2, AVX-512 (Tiger Lake microarchitecture)
	RAM	16 GB DDR4 3200 MT/s
Software & Runtime	Operating System	Windows 11 (10.0.26200.9168/25H2)
	Runtime Framework	.NET 10.0.12 (10.0.1226.42308, SDK 10.0.401)
	JIT Compiler	X64 RyuJIT x86-64-v4
	Garbage Collector	Concurrent Workstation GC
	Benchmarking Tool	BenchmarkDotNet v0.15.8
	Benchmark job parameters	IterationCount = 10, LaunchCount = 3, WarmupCount = 3
	Telemetry (AMD)	AMD uProf v5.3

The selection of these specific mobile architectures is highly intentional for evaluating AVX-512 energy efficiency. The AMD Zen 4 architecture introduced AVX-512 support without widening the physical execution datapaths to 512 bits. Instead, it executes 512-bit vectors across two 256-bit Floating-Point Units over two consecutive clock cycles (a technique known as double-pumping). While this mitigates severe thermal spikes and frequency penalties, it inherently alters the cycle-count expectations for vector instructions. Conversely, the Intel Tiger Lake microarchitecture is one of the few consumer mobile platforms to feature native 512-bit execution units, and it was selected as a contrast to double-pumped execution. As Section 3.6 shows, however, its client implementation provides a single 512-bit floating-point unit formed by fusing two 256-bit ports [9], so that its 512-bit issue capacity is, as on Zen 4, half its 256-bit capacity.

The benchmark suite was executed on both platforms under the runtime software configuration specified in the table. The marker-synchronised power-profiling harness was run on the AMD platform only. The RyuJIT compiler dynamically translates intermediate language into target-specific machine code at runtime. It maps .NET hardware intrinsics directly to the underlying CPU instruction sets (AVX2, AVX-512F, AVX-512BW).

The two platforms are not otherwise matched in hardware, and we do not claim that they are. While the software stack is identical, the machines differ in memory technology and capacity, frequency behaviour, core count, cache hierarchy and thermal design power, and are instrumented by different telemetry stacks. Absolute latencies are therefore not comparable across platforms and no such comparison is drawn. What each platform supports independently is the *within-platform* behaviour of the Vector256 and Vector512 variants under identical conditions on that machine. Both platforms yield latency and allocation under a common instrument. Power, and therefore energy, was measured on the AMD platform only.

Because the process targets x86-64-v4, AVX-512 encodings are available to the JIT for every benchmark in the suite, not only for the Vector512 variants. The scalar and Vector256 configurations are therefore not AVX-512-free baselines in the strict sense. At runtime `Vector512.IsHardwareAccelerated` reported `true` on both platforms, `Vector512<double>.Count` was 8 and `Vector<double>.Count` was 4, so the size-agnostic `Vector<T>` type remained 256 bits wide. This structural characteristic, alongside the lack of bare-metal telemetry for absolute cross-platform comparisons, constitutes a methodological limitation further addressed in Section 4.4.

2.5. Components Under Test

The benchmark suite evaluates the three stages of the parsing pipeline, systematically contrasting traditional scalar techniques with highly parallelized SIMD algorithms:

- **Lexical Analysis (Tokenization):** The conventional scalar `Tokenizer` evaluates input strings sequentially. This approach is inherently branch-heavy, utilizing cascading `if-else` or `switch` statements to classify characters (digits, operators, parentheses), which frequently stalls the CPU pipeline due to branch mispredictions. In contrast, the `SimdTokenizer` loads chunks of characters into wide vector registers (e.g., `Vector256`). By applying parallel byte-wise comparisons against known operator values, it generates a bitmask. The exact positions of operators are then rapidly extracted using hardware-accelerated bit-manipulation instructions such as `TrailingZeroCount` (TZCNT). This branchless classification substantially reduces pipeline flushes and keeps the execution ports saturated.
- **Infix-to-Postfix Conversion:** The classical Shunting Yard algorithm relies heavily on stack data structures. Stack-based operator-precedence parsing is a standard technique in compiler construction [10]. In managed environments, standard stack implementations require continuous heap tracking and dynamic resizing, contributing to GC pressure and poor spatial cache locality. To mitigate this, the `ScanBasedParser` employs a state-machine-free, scan-based strategy. By computing prefix sums or scanning flat arrays in place to determine evaluation order, it eliminates explicit stack manipulation, thereby maximizing L1 cache hits (a technique heavily utilized in SIMD-accelerated in-memory database systems [11]) and preventing unnecessary managed memory allocations.
- **Postfix Evaluation:** The baseline `PostfixNotationCalculator` processes Reverse Polish Notation sequentially, iterating through the token stream and resolving arithmetic operations one at a time. The proposed `SimdPostfixCalculator` holds each operand broadcast across all lanes of a `Vector256<double>` or `Vector512<double>` register, applies packed arithmetic to whole registers, and reads the result from lane zero. The lanes therefore carry identical copies, and the evaluator exploits no data-level parallelism. Vector width changes the encoding and register width of each operation, not its useful work. Their latency and reciprocal throughput at each vector width are characterized in Section 3.6. Throughput becomes the binding constraint only where independent operations can be issued concurrently, which is the regime that dense multi-expression workloads would enter.

To illustrate the elimination of branch misprediction penalties during lexical analysis, Algorithms 1 and 2 detail the core logic of the `SimdTokenizer`.

Algorithm 1 SIMD-Accelerated Tokenization (Part 1 of 2)

REQUIRE: S (ASCII math expression of length n)
ENSURE: T (Ordered list of extracted tokens: operands and operators)

IF AVX2 is not supported **THEN**
  RETURN ScalarTokenize(S)
END IF
$n \leftarrow \text{Length}(S)$
IF $n = 0$ **THEN**
  RETURN $\emptyset$
END IF
$T \leftarrow \emptyset$
// 1. Narrowing & Padding
$m \leftarrow \lceil n/32 \rceil \cdot 32$ *// Round up to 32*
$A[0 \ldots m-1] \leftarrow$ **new** ByteArray()
FOR $i \leftarrow 0$ **TO** $n-1$ **DO**
  $A[i] \leftarrow \text{ASCII}(S[i])$
END FOR
FOR $i \leftarrow n$ **TO** $m-1$ **DO**
  $A[i] \leftarrow$ ' ' *// Pad with space separators*
END FOR
$NumberStart \leftarrow -1$ *// Current operand start (-1 = none)*
// 2. Vectorized classification (per 32-byte chunk)
FOR $ChunkStart \leftarrow 0, 32, 64 \ldots m-32$ **DO**
  $Chunk \leftarrow \text{Load256}(A, ChunkStart)$
  // Parallel byte-wise equality tests (branchless, 32 lanes)
  FOR EACH $k \in \{+, -, *, /, (,)\}$ **DO**
    $CmpOp[k] \leftarrow \text{CompareEqualBytes}(Chunk, OpChar_k)$
  END FOR
  $CmpSpace \leftarrow \text{CompareEqualBytes}(Chunk, ' ')$
  $AnyOp \leftarrow \text{BitwiseOR}(CmpOp)$ *// Merge all operator masks*
  $OpMask \leftarrow \text{MoveMask}(AnyOp)$
  $SpaceMask \leftarrow \text{MoveMask}(CmpSpace)$
  $SepMask \leftarrow OpMask$ **OR** $SpaceMask$
  $ChunkEnd \leftarrow \text{Min}(32, n - ChunkStart)$
  IF $ChunkEnd < 32$ **THEN**
    $SepMask \leftarrow SepMask$ **AND** $(2^{ChunkEnd} - 1)$ *// Mask out padding*
  END IF
  // 3. Branchless bit-walk over separator mask
  $Remaining \leftarrow SepMask$
  $LocalPos \leftarrow 0$
  WHILE $Remaining \neq 0$ **DO**
    $BitPos \leftarrow \text{TrailingZeroCount}(Remaining)$
    IF $BitPos > LocalPos$ **AND** $NumberStart < 0$ **THEN**
      $NumberStart \leftarrow ChunkStart + LocalPos$
    END IF
    IF $NumberStart \geq 0$ **THEN**
      $AbsPos \leftarrow ChunkStart + BitPos$
      $\text{FlushOperand}(T, S, NumberStart, AbsPos - NumberStart)$
      $NumberStart \leftarrow -1$
    END IF
    IF bit $BitPos$ of $OpMask$ is set **THEN**
      $b \leftarrow A[ChunkStart + BitPos]$
      $T.\text{Add}(\text{OperatorToken}(\text{Lookup}[b]))$ *// Branchless lookup*
    END IF
    $LocalPos \leftarrow BitPos + 1$
    $Remaining \leftarrow Remaining$ **AND** $(Remaining - 1)$ *// Clear lowest set bit*
  END WHILE

Algorithm 2 SIMD-Accelerated Tokenization (Part 2 of 2)

IF $LocalPos < ChunkEnd$ **AND** $NumberStart < 0$ **THEN**
  $NumberStart \leftarrow ChunkStart + LocalPos$
END IF
END FOR

// 4. Flush trailing operand
IF $NumberStart \geq 0$ **THEN**
  FlushOperand$(T, S, NumberStart, n - NumberStart)$
END IF

RETURN T

PROCEDURE FlushOperand$(T, S, Start, Length)$:
  $Span \leftarrow S[Start \ldots Start + Length - 1]$
  $V \leftarrow$ ParseDouble$(Span)$

IF V is valid **THEN**
  T.Add(OperandToken(V))
ELSE
  THROW SyntaxError$(Span)$
END IF

As demonstrated in Algorithms 1 and 2, the `SimdTokenizer` implementation fundamentally restructures the parsing logic to exploit modern CPU instruction sets, achieving significant microarchitectural advantages:

- **Data-Level Parallelism:** Instead of a traditional scalar character-by-character loop, the algorithm processes 32-byte chunks simultaneously utilizing the `vpcmpeqb` (vector compare equal) instruction. Each iteration therefore consumes 32 input bytes rather than one, reducing the outer-loop iteration count by a factor of 32 relative to character-at-a-time scanning. The effect on total execution time is considerably smaller, because per-token processing and numeric conversion remain scalar. The measured latency advantage is 14.3% on AMD and larger on Intel (Sections 3.1 and 3.11).
- **Branchless Classification:** Conventional conditional branches (`if/else` or `switch` statements) are known to be detrimental to energy efficiency due to branch misprediction penalties. Here, they are largely replaced by bitwise logical OR operations (`vpor`) to construct token boundaries. The same vectorized classification strategy has been adopted by the two major browser engines for HTML scanning, where refinements of it yield a twenty-fold improvement over conventional scanning [12].
- **Bitmask Extraction:** The `vpmovmskb` (`MoveMask`) instruction condenses the 32-byte SIMD comparison result into a single 32-bit integer. This integer serves as a highly compact mathematical representation of all separator positions within the current chunk.
- **Hardware-Accelerated Bit Traversal (`tzcnt` + `blsr`):** Sequential character scanning is bypassed in favor of direct bitwise jumps. By leveraging TZCNT to find the next token and clearing the lowest set bit (`blsr`) to advance, the computational complexity scales strictly with the number of separators present, rather than the total string length. This data-parallel approach draws structural inspiration from high-throughput string processing, where replacing control flow with hardware-intrinsic data flow brings the instruction count down to a quarter or less of that of a conventional parser [13], and from vectorized integer decoding, where the same substitution governs the achieved rate [14]. The approach has since been carried over to standard-compliant URL parsing, where vectorized sub-algorithms cut the instruction count threefold relative to competing parsers [15]. Additionally, clearing the lowest set bit (`blsr`) to advance is a classical bit-manipulation idiom optimized in modern silicon [16].
- **Cross-Chunk Boundary Coherence:** The `NumberStart` variable persists across the 32-byte chunk iterations. This state-machine resilience guarantees correctness for

numerical operands spanning across arbitrary memory boundaries without requiring complex look-ahead or look-behind logic.

- **Architectural Fallback Strategy:** To ensure widespread compatibility across diverse hardware (e.g., older CPUs or ARM architectures lacking AVX-2), the algorithm gracefully delegates execution to the scalar tokenizer, ensuring correctness without compromising the fast-path performance.

Input Domain and Known Limitations

The vectorized fast path operates on single-byte ASCII. Input is narrowed to a byte array before classification, and as specified in Algorithm 1 the classification masks test only for the operator set $\{+, -, *, /, (,)\}$ and the space character. Three consequences follow and are stated here rather than left implicit.

Narrowing keeps only the low byte of each UTF-16 code unit. Most code points outside the ASCII range therefore narrow to a byte that is neither an operator nor a separator, are carried into the operand span, and are rejected during numeric conversion. This holds for typographic operator forms such as U+00D7, U+00F7 and U+2212, for full-width digits, and for non-breaking and ideographic spaces. It does not hold for code points whose low byte coincides with a recognized character. U+012A, U+012B, U+0128 and U+0129 narrow to `*`, `+`, `(` and `)`, and are accepted and evaluated without diagnostic, the parentheses by the SIMD path only. Inputs outside ASCII should therefore be rejected before narrowing, which a single vectorized comparison of the high bytes would achieve. Tab and line-feed characters are accepted by all evaluators, because they fall into the operand span and numeric conversion tolerates surrounding white space.

Unary operators are not supported. Because the sign characters appear in the separator set, a leading or embedded unary minus or plus is emitted as a binary operator and the expression is rejected. Scientific notation is likewise not supported. Numeric conversion does not accept an exponent, and the sign inside an exponent is in any case classified as an operator, splitting the literal. Transcendental functions and named constants are outside the grammar. Division by zero yields an IEEE 754 infinity rather than an error. Where AVX-2 is unavailable the implementation delegates to the scalar tokenizer, which shares the same input domain.

The SIMD evaluator holds its operand stack in a stack-allocated buffer of sixteen vector-width entries.

The padded buffer $A[0 \ldots m-1]$ constructed in Algorithm 1 is allocated per invocation and scales with input length. Section 3.7 shows that, beyond a threshold input size, the promotion cost of this buffer exceeds the instruction-count advantage that vectorization provides.

2.6. Benchmark Categories and Ablation Methodology

To systematically isolate and quantify the energy and latency impact of each component, the study follows a factorial ablation methodology. The benchmarks are partitioned into four logical categories using `[GroupBenchmarksBy(BenchmarkLogicalGroupRule.ByCategory)]`.

2.6.1. ParseOnly: Tokenization Isolation

The objective of this category is to isolate the cost of lexical analysis (string to token stream), independent of downstream conversion or evaluation operations. The specific benchmark scenarios evaluated for this isolated stage are outlined in Table 3.

Table 3. ParseOnly benchmark scenarios.

Benchmark	Description
`ScalarParse` (baseline)	Invokes traditional character-by-character scalar tokenization.
`SimdParse`	Invokes SIMD-based parsing to test the hypothesis that branchless character classification yields lower latency.

2.6.2. ConvertOnly: Infix-to-Postfix Conversion Isolation

This category isolates the algorithmic cost of converting a pre-tokenized infix stream to postfix (Reverse Polish Notation) format, utilizing pre-computed infix tokens to eliminate tokenization overhead from the measurement. The configurations tested during this conversion phase are detailed in Table 4.

Table 4. ConvertOnly benchmark scenarios.

Benchmark	Description
`ShuntingYardConvert` (base)	Classical stack-based Shunting Yard algorithm.
`ScanBasedConvert`	Scan-based conversion strategy, hypothesized to offer better cache locality and reduced branching relative to explicit stack push/pop operations.

2.6.3. EvalOnly: Postfix Evaluation Isolation

This category isolates the pure evaluation cost of a pre-computed postfix token stream, isolating the effect of vector width on evaluation without confounding parse or convert overhead. The evaluated postfix calculation scenarios are summarized in Table 5.

Table 5. EvalOnly benchmark scenarios.

Benchmark	Description
`ScalarEval` (baseline)	Sequential evaluation using standard scalar arithmetic loops.
`Vector256Eval`	Evaluation using 256-bit SIMD lanes (AVX-2).
`Vector512Eval`	Evaluation using 512-bit SIMD lanes (AVX-512).

2.6.4. FullPipeline: End-to-End Latency

This category measures the cumulative, real-world latency of the complete parse-convert-evaluate pipeline. This full factorial-style matrix enables the identification of interaction effects, such as whether SIMD tokenization gains are additive with SIMD evaluation gains, or whether pipeline stage transitions dominate the execution time. The complete matrix of these end-to-end scenarios is presented in Table 6.

The final row is a control rather than a competitor. The `System.Data.DataTable` engine provides substantially different functionality from a dedicated expression evaluator, so it bounds the comparison from above rather than contributing a configuration to the ablation. External evaluators are reported separately in Section 3.4 for the same reason.

Table 6. FullPipeline benchmark scenarios and configurations.

Benchmark	Tokenizer	Converter	Evaluator
`ScalarFullPipeline` (baseline)	Scalar	Shunting Yard	Scalar
`Vector256Full`	Scalar	Shunting Yard	Vector256
`Vector512Full`	Scalar	Shunting Yard	Vector512
`SimdTokenizerScalarFull`	SIMD	Shunting Yard	Scalar
`SimdFullPipeline_Vector256`	SIMD	Shunting Yard	Vector256
`SimdFullPipeline_Vector512`	SIMD	Shunting Yard	Vector512
`ScanBasedScalarFull`	SIMD	Scan-based	Scalar
`ScanBasedVector256Full`	SIMD	Scan-based	Vector256
`SortBasedDirectFull`	SIMD	Fused convert + evaluate pass	
`DataTableFull` (Control)	`System.Data.DataTable` reference engine		

3. Results

The comprehensive execution logs and raw hardware telemetry data supporting these microarchitectural findings are provided as supplementary materials and can be accessed via our Zenodo repository [17].

3.1. Isolated Pipeline Stages Analysis

To accurately quantify the microarchitectural behavior of the SIMD algorithms, the initial evaluation isolates each parsing stage. The empirical results are presented in Table 7.

Table 7. Performance and Allocation Metrics (AMD Zen 4). Latency was measured under BenchmarkDotNet, and allocation and Gen0 via ETW. Error denotes the half-width of the 99.9% confidence interval. Gen0 is reported per 1000 operations. Gen1 collections were not observed for any configuration in this table. The table reports the ablation of the proposed pipeline. Comparison against established .NET evaluators appears in Section 3.4. Energy is reported separately in Section 3.10, computed from power and latency measured within a single instrument.

Category	Method	Mean [ns]	Error [ns]	StdDev [ns]	Median [ns]	Ratio	Alloc. [B]	Gen0
ParseOnly	SimdParse	367.78	±9.50	13.31	361.20	0.86	1016	0.1211
	ScalarParse (Base)	428.92	±2.24	3.28	427.85	1.00	800	0.0954
ConvertOnly	ShuntingYard (Base)	224.92	±32.83	47.09	195.75	1.00	200	0.0238
	ScanBasedConvert	247.79	±6.18	9.06	246.28	1.10	192	0.0229
EvalOnly	Vector256Eval	69.30	±1.02	1.50	69.35	0.83	0	0.0000
	Vector512Eval	73.24	±1.49	2.18	73.84	0.88	0	0.0000
	ScalarEval (Base)	83.64	±1.44	2.11	83.95	1.00	144	0.0172
FullPipeline	SimdFullPipeline_Vector256	605.75	±5.53	7.92	605.68	0.92	1216	0.1450
	SimdFullPipeline_Vector512	623.23	±5.26	7.71	623.94	0.94	1216	0.1450
	Vector256Full	652.66	±4.28	6.13	652.45	0.99	1000	0.1192
	ScalarFullPipeline (Base)	661.51	±6.64	9.53	662.71	1.00	1144	0.1364
	Vector512Full	662.52	±7.53	10.79	662.57	1.00	1000	0.1192
	SortBasedDirectFull	682.70	±4.98	7.15	682.54	1.03	1832	0.2184
	SimdTokenizerScalarFull	687.82	±5.12	7.34	687.31	1.04	1400	0.1669
	ScanBasedVector256Full	706.37	±4.36	6.25	705.73	1.07	1208	0.1440
	ScanBasedScalarFull	723.12	±6.64	9.94	723.16	1.09	1352	0.1612

During the isolated evaluation phase (`EvalOnly`), the `Vector256Eval` implementation achieved the lowest latency, executing in 69.30 ns. The wider `Vector512Eval` benchmark showed a small but statistically robust regression, executing in 73.24 ns, a 5.7% increase (Welch's $t = 8.16$, $p = 7.7 \times 10^{-11}$, Hedges' $g = 2.08$). Section 3.6 shows that 512-bit

operations issue at half the rate of 256-bit ones on both platforms, and Section 4.1 discusses why this accounts only partly for the penalty, which is smaller on the Intel platform. Such behaviour is consistent with in-core throughput models, in which achievable performance is bounded by the execution ports and datapath widths actually present in a given microarchitecture rather than by the encoded vector width [18]. The energy consequence of this latency penalty is examined separately in Section 3.10, where power and latency are measured under a common instrument. Section 3.10 shows that in this regime the energy comparison is not resolved.

In the lexical analysis stage (`ParseOnly`), the `SimdParse` algorithm outperformed the baseline `ScalarParse` by 14.3% (367.78 ns versus 428.92 ns, $t = 24.43$, $p < 10^{-12}$, $g = 6.23$). The mechanism is the branchless classification described in Section 2.5, which replaces per-character conditional dispatch with vectorized mask generation. The gain is not free, since the SIMD tokenizer allocates 1016 B per operation against 800 B for the scalar path, a 27% increase, and correspondingly triggers 27% more Gen0 collections. Section 3.7 shows that this trade is favourable up to a bounded input size and reverses beyond it.

In the conversion stage (`ConvertOnly`), the baseline stack-based `ShuntingYardConvert` outperformed the `ScanBasedConvert` strategy at the reference length. The mean comparison (224.92 ns versus 247.79 ns) is significant after Holm–Bonferroni correction across the comparisons reported in this study ($t = 2.61$, $p = 0.014$, $g = 0.67$), but with a medium effect size and the weakest margin of any comparison in Section 3.9. The `ShuntingYardConvert` distribution is also strongly right-skewed (mean 224.92 ns, median 195.75 ns, standard deviation 47.09 ns), indicating intermittent environmental interference. On medians the separation is wider, at 195.75 ns versus 246.28 ns. The result at this length should therefore be read as established in direction but uncertain in magnitude. It becomes unambiguous under scaling, as reported in Section 3.3, where the gap reaches 32% at the 100× multiplier.

3.2. End-to-End Pipeline Performance

The combinatorial results of the complete execution pipeline (`FullPipeline`) synthesize the component-level analyses into a real-world evaluation context.

The most efficient configuration was `SimdFullPipeline_Vector256` at 605.75 ns, ahead of `SimdFullPipeline_Vector512` at 623.23 ns. The 2.9% separation between them ($t = 8.66$, $p = 4.9 \times 10^{-12}$, $g = 2.21$) is smaller than the 5.7% observed in isolation, consistent with a fixed per-evaluation penalty being diluted by the surrounding pipeline stages. Against the fully scalar baseline (`ScalarFullPipeline`, 661.51 ns), the best hybrid configuration improves end-to-end latency by 8.4%.

The factorial matrix does not decompose additively. Substituting the SIMD tokenizer alone degrades the pipeline (`SimdTokenizerScalarFull`, 687.82 ns, against 661.51 ns for the scalar baseline), while substituting the Vector256 evaluator alone improves it slightly (`Vector256Full`, 652.66 ns). Only the combination of both yields the observed 8.4% gain, indicating an interaction between tokenizer output layout and evaluator access pattern that neither substitution captures on its own. This interaction does not replicate on the Intel platform (Section 3.11).

The comparison is not uniformly favourable. Sections 3.4 and 3.5 situate these figures against established .NET evaluators and against the alternative of compiling the expression once to a delegate, which is the regime in which the proposed approach does not lead.

3.3. Algorithmic Scalability and Memory Pressure

To evaluate the asymptotic behavior of the parsing pipeline and examine the allocation behaviour of the pipeline under increased structural stress, the baseline expression was

subjected to a *Complexity Multiplier* scaling factor (1×, 10×, and 100×). Technically, this scaling is implemented via dynamic string concatenation, where the base expression is iteratively repeated and joined by the addition operator (i.e., `baseExpr + baseExpr + ...`). This linear horizontal expansion systematically stresses both the L1/L2 cache hierarchy and the managed heap. The results for the full pipeline executions are detailed in Table 8.

Table 8. Algorithmic Scalability and Memory Diagnostics (AMD Zen 4). Pipeline performance measured under increasing structural complexity multipliers (10× and 100×). The 99.9% CI and Median values are included to demonstrate execution stability under extreme memory pressure. Measured with the default BenchmarkDotNet job.

Mult.	Method	Mean [ns]	99.9% CI [ns]	Median [ns]	Allocated [B]	Gen0	Gen1
10×	SimdParse	5901.62	±116.73	5899.38	8560	1.0223	0.0229
	ScalarParse (Base)	7392.74	±147.00	7341.15	7784	0.9232	-
	ShuntingYard (Base)	2997.52	±57.41	2981.04	1208	0.1411	-
	ScanBasedConvert	3994.72	±79.14	4024.08	3720	0.4425	-
	Vector256Eval	1003.19	±19.93	1003.21	0	0.0000	-
	Vector512Eval	1043.08	±15.25	1043.73	0	0.0000	-
	ScalarEval (Base)	1578.95	±31.58	1581.06	1656	0.1965	-
	SimdFullPipeline_Vector256	10,031.36	±199.64	9992.41	9768	1.1597	0.0305
	SimdFullPipeline_Vector512	10,041.59	±200.04	10,017.03	9768	1.1597	0.0305
	SortBasedDirectFull	11,717.01	±233.52	11,742.47	18,328	2.1820	0.0458
	ScalarFullPipeline (Base)	11,062.48	±220.60	11,017.37	10,648	1.2665	0.0153
	DataTableFull	31,528.95	±862.15	30,490.54	47,352	5.6152	0.2441
100×	SimdParse	58,787.30	±1175.58	58,901.16	79,408	9.3994	1.8311
	ScalarParse (Base)	69,865.82	±1389.43	70,048.54	77,624	9.2773	0.6104
	ShuntingYard (Base)	29,270.54	±490.48	29,243.93	11,288	1.3428	-
	ScanBasedConvert	38,556.84	±764.85	38,662.34	36,120	4.2725	0.1221
	Vector256Eval	10,183.66	±192.99	10,139.38	0	0.0000	-
	Vector512Eval	10,262.79	±199.50	10,309.52	0	0.0000	-
	ScalarEval (Base)	15,586.80	±306.82	15,659.72	16,776	1.9836	-
	SimdFullPipeline_Vector256	97,178.00	±1811.23	96,972.52	90,696	10.7422	2.0752
	SimdFullPipeline_Vector512	98,892.42	±1531.56	99,028.91	90,696	10.7422	2.0752
	SortBasedDirectFull	116,688.75	±1736.25	116,253.35	161,488	19.2871	3.0518
	ScalarFullPipeline (Base)	111,490.62	±1368.57	111,529.60	105,688	12.5732	2.0752
	DataTableFull	293,951.84	±5851.96	293,775.07	387,200	45.8984	9.2773

The scaling data highlights the memory management challenges associated with non-optimized evaluators. At a 100× complexity multiplier, the standard framework `DataTableFull` allocates 387,200 Bytes per operation, resulting in 45.8984 Gen0 collections per 1000 operations. The hybrid `SimdFullPipeline_Vector512` limits allocations to 90,696 Bytes, a 76.58% reduction, and curtails Gen0 collections to 10.74. We note that this remains substantial allocation in absolute terms, because the zero-allocation property of this pipeline holds for the isolated evaluation stage, which receives pre-computed inputs, and does not extend to the complete pipeline at any scale.

The scaled results also refine the AVX-512 observation rather than simply confirming it. The `SimdFullPipeline_Vector256` variant remains ahead of its 512-bit counterpart at every multiplier, but the margin contracts as the workload grows. For the isolated evaluation stage it falls from 5.7% at 1× to 4.0% at 10× and 0.8% at 100×. This is the behaviour of a fixed per-evaluation cost rather than of a proportional throughput constraint. The reference expression performs a small, fixed number of vector operations per evaluation.

Whatever cost attaches to them is constant, and its share of total execution time falls as the token stream lengthens. Sections 3.6 and 4.1 examine its source, which the instruction-level measurements constrain but do not fully identify.

Finally, while the fused *SortBasedDirectFull* configuration remains competitive for baseline expressions, it scales less well under increased structural stress. At the 100× multiplier it runs 4.7% slower than the scalar baseline while allocating half as much again, and the allocation behaviour rather than the latency separation is the operative objection to this configuration.

3.4. Comparison Against Established Expression Evaluators

To situate the proposed pipeline among realistic alternatives, the suite was extended with three established .NET evaluators, each exercised in the usage modes a practitioner would adopt. Results appear in Table 9.

Table 9. Comparison against established .NET expression evaluators (AMD Zen 4). `EvalOnly` measures evaluation of a pre-prepared representation. `FullPipeline` measures a complete parse-and-evaluate cycle per invocation. The proposed configurations are repeated from Table 7 so that the ranking can be read without cross-reference.

Category	Method	Mean [ns]	StdDev [ns]	Alloc. [B]
EvalOnly	LinqExpressions_InvokePrecompiled	0.2151	0.0065	0
	NCalc_InvokeCompiledLambda	0.4352	0.0036	0
	Jace_InvokeCompiled	11.574	0.257	120
	Vector256Eval	69.297	1.496	0
	LinqExpressions_InvokeInterpreted	70.618	6.752	272
	Vector512Eval	73.241	2.182	0
	ScalarEval (Base)	83.637	2.107	144
	Jace_CalculateCached	143.835	5.675	888
	NCalc_EvaluatePrebuilt	260.611	17.021	1040
	NCalc_ParseAndEvaluate_Cached	349.557	17.986	1816
	DataTable_Compute_Hoisted	831.309	24.020	3800
ParseToEvaluable	LinqExpressions_BuildTree	380.65	4.43	624
	SimdParseAndConvert	560.91	5.30	1216
	ScalarParseAndConvert	609.09	21.82	1000
FullPipeline	SimdFullPipeline_Vector256	605.75	7.92	1216
	ScalarFullPipeline (Base)	661.51	9.53	1144
	Jace_ParseAndEvaluate_NoCache	1194.28	17.96	6136
	DataTable_Compute_PerCall	1206.95	32.60	8496
	DataTableFull (Control)	2634.31	78.37	13,368
	NCalc_ParseAndEvaluate	7009.12	300.80	13,368
	LinqExpressions_CompileAndInvoke	18,777.76	632.02	5200

Two results merit emphasis because they bound the contribution rather than support it. A delegate precompiled through `System.Linq.Expressions` evaluates the reference expression in 0.2151 ns, and a compiled NCalc lambda in 0.4352 ns. Both are some two orders of magnitude faster than any token-walking evaluator, our own included. The difference is structural rather than a deficiency of implementation. A compiled delegate has resolved the expression to straight-line machine code with constants folded, performing no token dispatch, whereas the proposed evaluator traverses a postfix token stream on every invocation. The comparison is not like-for-like, since the delegate cannot be obtained without a compilation step, but it establishes that the proposed evaluator is not competitive for repeated evaluation of a fixed expression. Within the class of interpretive evaluators, `Vector256Eval` is the fastest measured, marginally ahead of the interpreted LINQ expres-

sion path (69.30 ns against 70.62 ns) while allocating nothing against its 272 B. The latency ordering is specific to the AMD platform, whereas the allocation difference holds on both (Section 3.11).

Secondly, constructing a LINQ expression tree (380.65 ns) is faster than the proposed parse-and-convert path (560.91 ns) by 32%. The advantage of the proposed pipeline therefore rests on the complete parse-and-evaluate cycle rather than on any individual stage.

3.5. Amortisation of Preparation Cost

The preceding comparison measures a single evaluation. Because the two strategies distribute their cost differently, compilation pays a large fixed cost and a negligible marginal cost, while parsing pays a small fixed cost and a substantial marginal cost. The preferable strategy depends on how many times a given expression is evaluated. Table 10 reports the measurement.

Table 10. Total latency as a function of the number of evaluations of one expression (AMD Zen 4).

Method	N = 1	N = 10	N = 100	N = 1000
Linq_CompileThenInvoke [ns]	18,433.3	18,685.3	18,687.1	20,948.5
Simd_ParseOnceThenEval [ns]	618.1	1385.6	8483.1	81,107.8
Simd_ReparseEachTime [ns]	706.1	6211.3	64,741.4	640,994.6

Fitting $t(N) = a + bN$ by least squares yields a fixed cost of 18,514 ns and a marginal cost of 2.43 ns per evaluation for the compiled delegate, against 517 ns and 80.58 ns for parse-once-then-evaluate. The two curves intersect at $N \approx 230$. Below this threshold the proposed pipeline is the faster strategy, by a factor of thirty at a single evaluation. Above it, delegate compilation dominates and the margin grows without bound. We regard this threshold, rather than any single speedup ratio, as the operative result of the comparison, and it delimits the deployment context in which the proposed pipeline is appropriate, namely expressions supplied dynamically and evaluated a small number of times, rather than fixed expressions evaluated in a hot loop. The threshold lies higher on the Intel platform (Section 3.11).

The third series quantifies the value of separating parsing from evaluation. Re-parsing on every invocation costs 640.88 ns per evaluation, eight times the marginal cost of parsing once, which is the architectural separation the proposed pipeline is designed to exploit.

3.6. Instruction-Level Characterization of Vector Width

The evaluation benchmarks conflate two effects, namely the cost of the 512-bit encoding itself, and the absence of data-level parallelism in the evaluator, which broadcasts each operand across all lanes. To separate them, a dedicated microbenchmark measures identical arithmetic at 128, 256 and 512 bits with a distinct value in every lane, in three configurations: a dependent chain measuring instruction latency, an independent stream measuring reciprocal throughput, and a per-element normalization measuring realized width scaling. Results for both platforms appear in Table 11. Cycle figures for AMD are derived from the measured execution times and the effective core frequency recorded by AMD uProf during each segment (Section 3.10), not from performance-counter events. No frequency telemetry is available for the Intel platform. There, the quotient of latency and reciprocal throughput, which does not depend on clock frequency, stands in for cycle figures.

Table 11. Instruction-level characterization of vector width on both platforms. Latency is measured on a dependent chain and throughput on an independent stream. Per-element cost normalizes throughput by lane count. For AMD, cycle equivalents use the effective core frequency measured during each segment by AMD uProf, 4920 MHz for the 256-bit kernel and 4973 MHz for the 512-bit kernel (Section 3.10). The scalar and Vector128 configurations were not profiled separately, and their reciprocal throughput is converted at the 256-bit frequency. This is adequate for the ordering shown but should not be read as an independently measured cycle count. No frequency telemetry was recorded on the Intel platform, and no cycle equivalents are given for it. The quotient of latency and reciprocal throughput does not depend on clock frequency. It equals the number of independent operations that must be in flight to saturate the execution units.

Configuration	Scalar	Vector128	Vector256	Vector512	V512/V256
AMD Zen 4 (Ryzen 7 PRO 7840U)					
Latency [ns]	0.5990	0.5991	0.6037	0.5989	0.99
Latency [cycles]	—	—	2.97	2.98	—
Recip. throughput [ns]	0.0532	0.0527	0.0527	0.1022	1.94
Recip. throughput [cycles]	0.26	0.26	0.26	0.51	—
FMA throughput [ns]	—	—	0.1022	0.2033	1.99
FMA throughput [cycles]	—	—	0.50	1.01	—
Latency/recip. throughput	11.3	11.4	11.5	5.9	0.51
Per-element cost [ns]	0.0534	0.0262	0.0132	0.0128	0.97
Speedup vs. scalar	1.00×	2.04×	4.05×	4.17×	—
Intel Tiger Lake (Core i5-11400H)					
Latency [ns]	0.8863	0.8864	0.8874	0.9080	1.02
Recip. throughput [ns]	0.1118	0.1118	0.1118	0.2289	2.05
FMA throughput [ns]	—	—	0.1143	0.2360	2.06
Latency/recip. throughput	7.9	7.9	7.9	4.0	0.50
Per-element cost [ns]	0.1117	0.0559	0.0282	0.0286	1.01
Speedup vs. scalar	1.00×	2.00×	3.96×	3.91×	—

Four observations follow.

Instruction latency is essentially invariant with vector width. On AMD the four configurations agree to within 0.8%, and the cycle equivalent is consistent with the three-cycle latency documented for Zen 4 packed floating-point addition and multiplication [19]. On Intel the scalar, Vector128 and Vector256 configurations agree to within 0.1%, while Vector512 is slightly slower. That excess is matched by a similar excess of the 512-bit throughput ratios over 2.00, so the frequency-independent quotient is unaffected. Section 4.1 discusses its likely origin. On both platforms a dependency-chained evaluator is governed by this latency, and it does not improve with wider registers.

Reciprocal throughput doubles at 512 bits on both platforms. On AMD a 512-bit FMA occupies one cycle where a 256-bit FMA occupies half of one, which is what results when a single 512-bit operation is issued as two micro-operations across the paired 256-bit datapaths [19,20]. Intel reaches the same ratio by a different mechanism. On the client Tiger Lake core, 512-bit floating-point operations execute on a single 512-bit unit formed by fusing the two 256-bit ports, the second 512-bit FMA unit being present only on server implementations [9]. On both platforms a 512-bit operation consumes the issue capacity of two 256-bit operations.

The latency–throughput quotient agrees with documented figures on both platforms. The Intel values are consistent with a four-cycle latency and reciprocal throughputs of 0.5 and 1.0 cycles [9]. On both platforms the quotient halves at 512 bits, because each 512-bit operation carries twice the work but issues at half the rate, so half as many need be in flight to saturate the execution units.

Realized width scaling ceases above 256 bits on both platforms. Per-element cost falls cleanly from scalar to 128 and 256 bits, then stalls. On AMD only 3.1% of the width scaling available between 256 and 512 bits is realized, and on Intel none is, with the 512-bit path being marginally slower per element. Because these kernels carry a distinct value in every lane, the effect is not an artifact of the seven-operand reference expression.

The doubling of reciprocal throughput is therefore not a signature unique to double-pumped execution, but of any core whose 512-bit issue capacity is half its 256-bit issue capacity.

These measurements account for the absence of any per-element gain at 512 bits, but only partly for the evaluation-level penalty reported in Sections 3.1 and 3.3. Section 4.1 examines why. The finding that does not depend on this question is the one relevant to design. On neither platform does the 512-bit path deliver more work per unit time than the 256-bit path.

3.7. Input Length and Nesting Depth

The scaling experiment of Section 3.3 varies workload size by concatenation, which increases token count but not structural diversity. Two further sweeps separate these factors: one varies input length at fixed nesting depth, the other varies nesting depth at fixed length. Results appear in Tables 12 and 13.

Table 12. Length sweep at fixed nesting depth 4 (AMD Zen 4). Ratio is SIMD relative to scalar, so values below unity favour SIMD. Allocation and generational counts are reported for the tokenization stage in isolation, which is where the two paths differ. Gen counts are per 1000 operations and a dash denotes a generation not reached. Full-pipeline allocation runs in the opposite direction and is discussed in the text.

	Full Pipeline			Parse Stage			Parse-Stage Memory			
Case	**Scalar [ns]**	**SIMD [ns]**	**Ratio**	**Scalar [ns]**	**SIMD [ns]**	**Ratio**	**Scalar [KB]**	**SIMD [KB]**	**Scalar Gen0/1/2**	**SIMD Gen0/1/2**
L40	566.2	547.1	0.97	402.0	330.5	0.82	0.74	0.99	0.09/–/–	0.12/–/–
L131	1840.2	1597.5	0.87	1218.9	926.8	0.76	1.96	2.13	0.24/–/–	0.26/–/–
L517	7477.8	6363.2	0.85	4683.5	3446.7	0.74	7.12	8.30	0.87/–/–	1.01/0.02/–
L2050	29,557.6	25,853.3	0.87	18,043.2	13,665.9	0.76	27.65	32.12	3.36/0.03/–	3.94/0.35/–
L8196	125,399.2	100,673.4	0.80	78,815.5	54,869.2	0.70	109.96	127.32	13.43/0.98/–	15.56/3.85/–
L32774	544,378.6	616,718.2	1.13	332,233.8	368,849.5	1.11	439.12	507.96	53.22/16.60/–	83.01/82.52/41.50

The advantage of vectorized tokenization is bounded above as well as below. It builds to a maximum at L8196, where the SIMD pipeline runs at 0.80 of scalar latency, and then reverses. At L32774 it runs at 1.13, that is 13% slower. The memory diagnostics locate the cause in promotion rather than in allocation volume. The SIMD tokenizer allocates more than the scalar tokenizer at every length measured, as Tables 7 and 8 already show at the reference expression and under scaling, but the excess is near-constant at 16–17% from L517 onward (Table 12) and therefore cannot by itself account for a change of sign. What changes at L32774 is the fate of those allocations. The per-invocation padded buffer of Algorithm 1 scales with input length, and at this size nearly every Gen0 collection on the SIMD path is accompanied by a Gen1 collection (82.52 against 83.01 per 1000 operations), against under a third on the scalar path (16.60 against 53.22). The SIMD path alone reaches Gen2, at 41.50 collections per 1000 operations, where the scalar path reaches it at no length measured. The separation is visible earlier in milder form, as the SIMD parse stage reaches Gen1 at L517, where the scalar path does not reach it until L2050. The crossover in latency coincides with the first appearance of Gen2. This bounds the applicability of the approach

and identifies buffer pooling as the necessary next step, since a pooled buffer would not be promoted at all.

Two qualifications. The figures above are for the tokenization stage in isolation. Across the complete pipeline the SIMD configuration allocates less than the scalar one at every length beyond L40, by 5–9% (full-pipeline figures in the supplementary logs, and at L40 itself the ordering matches that of the reference expression in Table 7), so the tokenizer's allocation overhead is more than offset downstream and the crossover is a promotion effect confined to one stage rather than a property of the pipeline's total footprint. The crossover also runs opposite to the direction anticipated in Section 4.3 for the conversion stage. There a large working set was expected to favour the flat-array strategy, whereas here a large working set penalises the vectorized path, and in both cases the operative variable is allocation behaviour rather than cache capacity.

Table 13. Nesting-depth sweep at fixed input length near 258 characters (AMD Zen 4), right-associated nesting. Left-associated results were indistinguishable and are omitted. Peak operand-stack depth is $D + 1$ for right-associated and 2 for left-associated nesting at every depth D.

Depth	Scalar Full [µs]	SIMD Full [µs]	Ratio	Scalar Alloc. [KB]	SIMD Alloc. [KB]
d1	3.661	3.252	0.89	5.06	4.56
d2	3.733	3.270	0.88	5.13	4.61
d4	3.614	3.129	0.87	5.12	4.64
d8	3.532	3.125	0.88	5.23	4.77
d15	3.647	3.288	0.90	5.41	4.97

Latency is flat across nesting depth. Scalar latency spans 3.532–3.733 µs, a 5.7% range with no monotonic trend, and SIMD latency spans 3.125–3.288 µs, a 5.2% range. The ratio between them stays within 0.87–0.90. Nesting direction has no measurable effect. Allocation is the only quantity that tracks depth, rising 6.9% from d1 to d15 as the operator stack deepens. For expressions of this length, latency is therefore governed by token count rather than by structural depth, at least up to depth 15. We do not extrapolate beyond that bound, which also coincides with the operand-stack capacity of the SIMD evaluator (Section 2.5). The divergence anticipated in Section 4.3 would require nesting deep enough to exceed cache capacity, which this sweep does not approach.

3.8. Rejection of Malformed Input

Correct inputs are only part of a parser's workload. Table 14 reports rejection latency across twelve malformed inputs, ordered by the position at which the defect becomes detectable.

Table 14. Rejection latency for malformed inputs (AMD Zen 4). Ratio is SIMD relative to scalar.

Case	Scalar [µs]	SIMD [µs]	Ratio	Defect Position
bad-empty	1.207	1.675	1.39	position 0
bad-spaces	1.201	1.708	1.42	position 0
bad-emptypar	1.604	1.786	1.11	early
bad-letters	1.580	1.728	1.09	early
bad-bareexp	1.485	1.620	1.09	mid
bad-hugeexp	1.486	1.619	1.09	mid
bad-dbldot	1.502	1.626	1.08	mid
bad-tripart	1.605	1.650	1.04	mid
bad-dblop	2.146	1.949	0.91	late
bad-unbal-l	1.688	1.591	0.94	end of scan
bad-trailop	2.368	1.991	0.84	end of input
bad-unbal-r	1.738	1.325	0.76	end of input

Rejection latency is dominated by exception handling, and in five of the twelve cases (`bad-empty`, `bad-spaces`, `bad-emptypar`, `bad-dblop` and `bad-trailop`) the SIMD path rejects through an internal index error rather than the syntax error raised by the scalar path. Those five ratios compare different exception paths and are not attributable to the tokenizer. This includes both position-zero cases, and for empty input the SIMD tokenizer returns before any narrowing. Among the seven cases in which both paths raise the same syntax error, the ordering follows the structure of the vectorized path. Narrowing and padding the input buffer, then loading the first chunk, constitute a fixed setup cost that the scalar tokenizer avoids by terminating at the first offending character, and the SIMD path runs 4–9% slower where the defect is detectable early or mid-input. Where the input must be scanned to its end before the defect becomes detectable, the setup amortizes and the SIMD path is up to 24% faster. Raising a syntax error before the operand stack is read would align the two paths.

Correctness was checked on a separate corpus of 69 inputs combining the length and depth sweeps with functions, unary operators, scientific notation, Unicode and white-space variants and the malformed inputs above. On every one of the 23 inputs that all three evaluators accept, the scalar, Vector256 and Vector512 results are bit-identical, a difference of zero units in the last place, with no spread across lanes. The remaining inputs expose the limitations stated in Section 2.5.

3.9. Statistical Significance and Effect Sizes

Several comparisons in this study involve differences of a few percent, where confidence intervals alone do not establish whether an effect is present. Table 15 reports Welch's two-sample t-test and Hedges' g for the principal comparisons, computed on $n = 30$ measurements per configuration.

Table 15. Significance and effect sizes for the principal comparisons (AMD Zen 4). Welch's two-sample t-test, $n = 30$ per arm. Here g is Hedges' g with small-sample correction.

Comparison	Δ [ns]	Δ%	t	p	g
Vector512Eval vs. Vector256Eval	+3.94	+5.69	8.16	7.7×10^{-11}	2.08
ScalarEval vs. Vector256Eval	+14.34	+20.69	30.39	$<10^{-12}$	7.75
ScalarEval vs. Vector512Eval	+10.40	+14.20	18.77	$<10^{-12}$	4.78
ScanBasedConvert vs. ShuntingYard	+22.86	+10.16	2.61	0.014	0.67
ScalarParse vs. SimdParse	+61.15	+16.63	24.43	$<10^{-12}$	6.23
SimdFull_V512 vs. SimdFull_V256	+17.48	+2.89	8.66	4.9×10^{-12}	2.21
ScalarFullPipeline vs. SimdFull_V256	+55.76	+9.20	24.65	$<10^{-12}$	6.28
Vector256Full vs. SimdFull_V256	+46.91	+7.74	25.64	$<10^{-12}$	6.53
SimdTokenizerScalarFull vs. ScalarFullPipeline	+26.31	+3.98	11.98	$<10^{-12}$	3.05
Fma_V512 vs. Fma_V256	+0.1011	+98.92	375.05	$<10^{-12}$	95.58
Throughput_V512 vs. V256	+0.0495	+93.93	294.07	$<10^{-12}$	74.94
Latency_V512 vs. Latency_V256	−0.0048	−0.80	−2.97	0.005	−0.76

The two comparisons involving small margins, Vector512 against Vector256 in isolation and end to end, both clear significance with large effect sizes. We characterize them as statistically robust but practically small, and both descriptions are material. The effect is real, and it is a few percent.

Two entries require qualification. The `ConvertOnly` comparison at the reference length reaches $p = 0.014$ with a medium effect size. It survives Holm–Bonferroni correction across the twelve comparisons reported here, but only as the last step of the sequence, where the threshold is α itself, and its baseline distribution is strongly right-skewed. It is discussed in Section 3.1. The latency comparison between vector widths also survives correction, but the

difference is −0.0048 ns, approximately 0.02 cycles, and negative in sign. At this precision statistical significance carries no physical content, and the correct reading is equivalence within 1%.

3.10. Direct Power Characterization

Energy cannot be derived from latency alone without an assumption about power. To measure both under a common instrument, the principal configurations were re-executed under AMD uProf in the sustained harness described in Section 2.1, sampling package power, per-core power and per-thread effective frequency at a nominal 100 ms interval (median 102 ms, measured from the record timestamps) (Table 16). Two seconds were trimmed from each end of every segment to exclude transitions, leaving approximately 404 samples per segment, and the idle windows provide the baseline against which dynamic energy is computed. Section 3.10.3 reports the ordering control applied to this session.

Table 16. Per-variant power and effective frequency (AMD Zen 4). Dynamic core power is the measured sum over cores less the idle baseline of 2.94 W. The idle baseline is the combined leading and trailing windows ($n = 208$), which agree to 0.03 W eight minutes apart. Values are from the shuffled ordering.

Segment	Package [W]	Cores [W]	Dyn. Core [W]	Max Core [W]	Eff. Freq [MHz]
ScalarEval	22.18 ± 2.72	11.96	9.02	9.56	4846
Vector256Eval	21.27 ± 3.21	11.89	8.95	9.22	4901
Vector512Eval	20.63 ± 2.59	10.88	7.94	8.96	4926
ScalarFullPipeline	21.57 ± 2.56	12.18	9.24	10.16	4855
SimdFullPipeline_Vector256	21.87 ± 2.34	12.33	9.40	10.38	4884
DataTableFull	23.65 ± 3.86	19.53	16.59	15.09	4877
Throughput_Vector256	22.32 ± 2.60	19.08	16.14	17.12	4920
Throughput_Vector512	21.52 ± 2.60	15.61	12.68	13.37	4973
Idle (baseline)	6.33 ± 2.56	2.94	—	1.07	—

Two findings follow.

Power is not constant across algorithmic variants. Package draw spans 20.63 to 22.18 W across the three evaluation variants and 20.63 to 23.65 W across all measured segments. The energy figures reported here are computed as the dynamic term above the measured idle baseline rather than as a uniform product.

The wider vector path draws less power, not more, in the isolated throughput kernel, and does so in all three segment orderings. In the evaluation workload only the shuffled ordering shows the same direction, so no power ordering is established there. Effective core frequency is marginally higher during the 512-bit segments than during their 256-bit counterparts or the scalar baseline, so no thermal intervention is present. Section 4.1 examines the mechanism behind both observations.

Energy per operation (E) and the EDP are computed from the two quantities measured above, the dynamic core power of Table 16 and the execution latency recorded in the same session:

$$E = P \times t \tag{2}$$

$$EDP = E \times t = P \times t^2 \tag{3}$$

where P is the dynamic core power in Watts (W), that is the summed per-core draw less the measured idle baseline, t is the latency in seconds converted from the empirical nanoseconds, E is the energy per operation in Joules (practically evaluated in microjoules,

μJ), and EDP is expressed in Joule-seconds (J·s). The resulting values are reported in Table 17.

Table 17. Measured energy per operation (AMD Zen 4). Dynamic core power and execution latency are both taken from the profiled session, so each energy figure is the product of two quantities measured under the same instrument. Latencies differ from Table 7 because the two instruments run under different configurations. That suite measures short isolated runs under the default runtime settings, whereas this harness measures sustained execution with tiered compilation disabled and the process pinned (Section 2.1).

Configuration	P_{dyn} [W]	t [ns]	E_{dyn} [μJ]	EDP [10^{-14} J·s]
ScalarEval (Base)	9.02	281.46	2.539	71.45
Vector256Eval	8.95	83.55	0.748	6.25
Vector512Eval	7.94	96.86	0.769	7.45
ScalarFullPipeline (Base)	9.24	1396.83	12.907	1802.9
SimdFullPipeline_Vector256	9.40	957.70	9.002	862.1
DataTableFull (Control)	16.59	4750.83	78.816	37,440.0

3.10.1. Energy in Two Regimes

Combining measured power with measured rate under a common instrument (Table 17) separates two regimes that latency figures alone conflate.

In the isolated throughput kernel, where lanes are fully occupied and operations are independent, the 512-bit path delivers 78.13 Gelement/s against 75.76 Gelement/s while drawing 12.68 W against 16.14 W. Energy per element is therefore 162.3 pJ against 213.0 pJ, a 23.8% saving, with a 26.1% lower energy-delay product. In this regime the wider register is the more energy-efficient choice despite offering only a 3.1% throughput advantage. The rate figures underlying this comparison replicate across all three segment orderings to within 1.1%, and the energy saving per element is reproduced in all three orderings.

In the evaluation workload the picture differs, because each operation yields a single useful result and the operations form a dependency chain. Here, in the shuffled ordering, the 512-bit path draws 7.94 W against 8.95 W while taking longer, giving 0.769 μJ against 0.748 μJ per operation and energy-delay products of 7.45×10^{-14} against 6.25×10^{-14} J · s. We report this comparison with an explicit reservation. The `Vector256Eval` segment exhibited 48.7% variation across the three orderings (Section 3.10.3), and substituting its cross-ordering mean in place of the profiled value reverses the sign of the energy difference. The latency comparison for these two configurations, measured under BenchmarkDotNet with narrow confidence intervals, is the reliable one. The energy comparison in this regime should be treated as indicative pending a more stable measurement of that configuration.

The comparisons that do not involve that configuration are unaffected. Against the scalar baseline, the Vector256 evaluator consumes 70.5% less dynamic energy per operation and yields a 91.3% lower energy-delay product. At the pipeline level the SIMD configuration consumes 9.002 μJ against 12.907 μJ for the scalar baseline, a 30.3% reduction, and 78.816 μJ for the `DataTable` control, a factor of 8.76. In energy-delay terms these become reductions of 52.2% and a factor of 43.4, respectively.

3.10.2. Race-to-Sleep

The residency benefit asserted earlier in this work was tested directly under a fixed 10 ms duty cycle, in which each period performs a fixed quantum of work and then sleeps for the remainder. Holding delivered work constant isolates the fraction of wall-clock time spent in the active C0 state.

The two arms deliver the same work to within 0.05% (Table 18). Active residency falls from 69.07% to 51.62%, a relative reduction of 25.3%, and idle residency rises by

17.45 percentage points. The power trace of this run records package energy of 1052 J for the scalar arm and 886 J for the SIMD arm over 60 s, a reduction of 15.9% for identical delivered work. The trace records power but not C-state depth, so how the saving divides between active and idle intervals is not established here.

Table 18. Duty-cycle experiment, 10 ms period, 60 s per arm (AMD Zen 4).

	ScalarFullPipeline	SimdFullPipeline_Vector256
Wall time [s]	60.003	60.003
Operations delivered	28,780,774	28,765,936
Latency per operation [ns]	1439.98	1076.78
Busy time [s]	41.44	30.97
Idle time [s]	18.56	29.03
Active (C0) residency	69.07%	51.62%

3.10.3. Ordering Control

Segment position does not systematically affect the measurements (Table 19). `ScalarEval` yields 276.86 ns in first position and 278.92 ns in last, and the isolated throughput kernels vary by less than 1.1% regardless of position. Thermal drift and warm-up are therefore not confounding factors in this suite. The exception is `Vector256Eval`, which varies by 48.7% across the three orderings without ordering by position, indicating a source of variance internal to the benchmark rather than to its placement.

Table 19. Cross-ordering consistency, ns per operation, with segment position in the run given in brackets.

Segment	Forward	Reverse	Shuffled	Mean	Spread
ScalarEval	276.86 [1]	278.92 [8]	281.46 [1]	279.08	1.7%
Vector256Eval	124.22 [2]	93.12 [7]	83.55 [4]	100.30	48.7%
Vector512Eval	96.25 [3]	88.02 [6]	96.86 [5]	93.71	10.1%
ScalarFullPipeline	1406.67 [4]	1405.39 [5]	1396.83 [3]	1402.96	0.7%
SimdFullPipeline_Vector256	972.07 [5]	983.36 [4]	957.70 [6]	971.04	2.7%
DataTableFull	4527.90 [6]	4712.09 [3]	4750.83 [8]	4663.61	4.9%
Throughput_Vector256	0.0534 [7]	0.0529 [2]	0.0529 [2]	0.0530	1.1%
Throughput_Vector512	0.1027 [8]	0.1021 [1]	0.1025 [7]	0.1024	0.6%

The sustained-throughput harness also reports a larger full-pipeline advantage than BenchmarkDotNet. The SIMD pipeline runs at 0.692 of scalar latency here, the ratio spanning 0.686 to 0.700 across the three orderings, against 0.92 under BenchmarkDotNet. The two harnesses measure different quantities. BenchmarkDotNet subtracts invocation overhead and measures short isolated runs, whereas this suite measures sustained execution with warm caches and trained branch predictors, and we report both rather than selecting between them.

3.11. Replication on Intel Tiger Lake

The latency and allocation suite was repeated on the Intel Core i5-11400H under the same BenchmarkDotNet job parameters. Because absolute latencies are not comparable across the two machines (Section 2.4), the results are summarized in Table 20 as ratios within each platform. Allocation per operation agrees between the two platforms to within 1 B for every configuration reported in Tables 7 and 9, so the allocation figures given there apply to both.

Table 20. Within-platform ratios for the principal comparisons. Each entry is the ratio of BenchmarkDotNet means measured on one machine (30 measurements per configuration, 10 iterations across 3 launches). Values below unity mean the first-named configuration is faster. Absolute latencies are not compared across platforms (Section 2.4). Depth-sweep ranges are for right-associated nesting, as in Table 13.

Comparison	AMD Zen 4	Intel Tiger Lake
Isolated stages		
SimdParse/ScalarParse	0.86	0.76
ScanBasedConvert/ShuntingYard	1.10	1.19
Vector256Eval/ScalarEval	0.83	0.79
Vector512Eval/Vector256Eval	1.057	1.010
Full pipeline		
SimdFull_V256/ScalarFull	0.92	0.80
SimdFull_V512/SimdFull_V256	1.029	1.014
SimdTokenizerScalarFull/ScalarFull	1.04	0.92
Vector256Full/ScalarFull	0.99	0.90
DataTableFull/SimdFull_V256	4.35	4.37
Against established evaluators		
LinqInterpreted/Vector256Eval	1.02	0.90
SimdParseAndConvert/Linq_BuildTree	1.47	1.37
Thresholds and sweeps		
Amortisation crossover N	≈230	≈310
Length sweep, full pipeline, L8196	0.80	0.79
Length sweep, full pipeline, L32774	1.13	1.03
Length sweep, parse stage, L32774	1.11	1.05
Depth sweep, full pipeline, d1–d15	0.87–0.90	0.82–0.84
Rejection, SIMD/scalar, 12 inputs	0.76–1.42	0.77–1.43

Every isolated-stage comparison replicates in direction, and the SIMD advantage is larger on Intel for both tokenization and the complete pipeline. The ConvertOnly comparison, which on AMD was the weakest comparison at the reference length and strongly right-skewed (Section 3.1), is unambiguous on Intel, where the stack-based baseline shows a standard deviation of 5.3 ns on a mean of 242.8 ns.

The thresholds replicate as well. Vectorized tokenization inverts at the same input length, and the SIMD path is again the only one to reach Gen2 there, at 41.5 collections per 1000 operations in the parse stage. Latency is flat across nesting depth, and rejection latency follows the same ordering by defect position, each of the twelve ratios lying within 0.04 of its AMD counterpart. The amortisation crossover moves to a higher N. The least-squares fit gives a fixed cost of 25,785 ns and a marginal cost of 0.62 ns per evaluation for the compiled delegate, against 741 ns and 80.76 ns for parse-once-then-evaluate. The marginal cost of the proposed pipeline is thus nearly unchanged from AMD (Section 3.5), and the shift arises from the higher fixed cost of delegate compilation.

Three AMD results do not replicate. First, the non-additive interaction reported in Section 3.2 is specific to AMD. On Intel each substitution alone improves the pipeline, and their combination is close to the product of the two individual ratios (0.84). Second, the interpreted LINQ path is faster than `Vector256Eval` on Intel, so the ranking among interpretive evaluators reported in Section 3.4 is platform-specific. Third, the Vector512 penalty is smaller, both in isolation ($t = 2.61$, $p = 0.012$, $g = 0.66$) and end to end ($t = 3.58$, $p = 7 \times 10^{-4}$, $g = 0.91$), with effect sizes well below the corresponding AMD values in Table 15. Section 4.1 examines why.

4. Discussion

The primary objective of this study was to critically evaluate the energy and latency characteristics of mathematical expression parsing in MREs, specifically contrasting scalar state-dependent algorithms with SIMD-accelerated pipelines. The empirical results demonstrate that while vectorization generally yields substantial throughput improvements, naive utilization of the widest available SIMD registers does not uniformly guarantee optimal energy efficiency.

4.1. The AVX-512 Paradox: Halved Issue Rate Rather than Frequency Downclocking

A central finding of this analysis is the performance regression observed when transitioning from AVX-2 (Vector256) to AVX-512 (Vector512) instructions. On the AMD Zen 4 mobile platform, despite AVX-512 nominally doubling the data-level parallelism available per instruction, the `Vector512Eval` benchmark required 5.7% more execution time than its 256-bit counterpart (Table 7), a difference that is statistically robust but modest in magnitude. On the Intel Tiger Lake platform the regression is smaller, with an effect size roughly a third of that on AMD (Section 3.11). On neither platform does the wider register reduce latency.

The instruction-level measurements of Section 3.6 identify what the two platforms share. Packed floating-point latency is essentially invariant with vector width, while reciprocal throughput doubles at 512 bits and per-element throughput improves marginally or not at all (Table 11). The mechanisms differ, double-pumping across paired 256-bit datapaths on Zen 4 [19,20] and a single port-fused 512-bit unit on the client Tiger Lake core [9], but the consequence for software is the same. A 512-bit operation consumes the issue capacity of two 256-bit operations. The contrast the study was designed to test, double-pumped against native execution, thus resolves into a common constraint rather than a difference, and the instruction-level data do not isolate double-pumping as such. We attribute the halved rate to issue capacity rather than to decode cost or port contention on the strength of the documented figures, since the present measurements include no counters that would separate these.

The halved issue rate explains why 512 bits bring no per-element gain, but not by itself why the evaluation-level penalty is several times larger on AMD than on Intel. A stack-carried dependency chain leaves issue slots idle, so a halved issue rate should cost little, which is what the Intel measurement shows. The larger AMD penalty, and its contraction with expression length (Section 3.3), indicate a fixed per-evaluation cost specific to that platform's 512-bit code path. The evaluator performs no horizontal reduction and reads its result from lane zero, so the candidates are operations it performs on every token and the microbenchmark excludes by design, namely the 512-bit broadcast of each operand and the store and reload of 64-byte stack entries through which the dependency chain passes. We leave their identification to counter-based measurement (Section 4.4).

The direction of the difference is opposite to what the frequency-offset literature on native 512-bit silicon would suggest. On AMD, Vector512 draws less power than Vector256, not more, in the isolated throughput kernel, consistently across all three segment orderings (Table 16). This follows from the halved issue rate, which lowers switching activity per unit time. The effective core frequency confirms the absence of any thermal intervention (Figure 1), being marginally higher during the 512-bit segments than during their 256-bit counterparts and the scalar baseline. On this microarchitecture, wide vector execution neither spikes power nor triggers downclocking. AMD's published description of this core, as deployed in the fourth-generation EPYC parts, presents AVX-512 on Zen 4 as a power-efficiency feature rather than a widening of the physical datapath, which is consistent

with the frequency behaviour measured here on the mobile implementation of the same core [20].

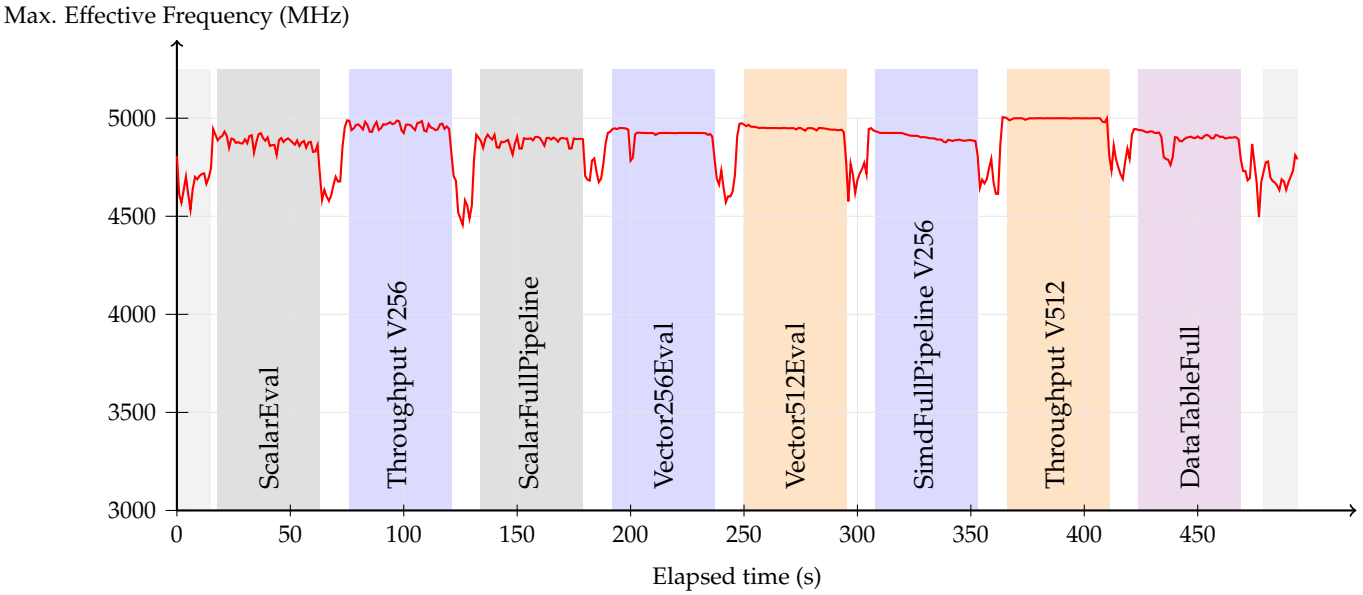


Figure 1. Effective frequency of the peak active core on the AMD Zen 4 platform during the profiled session (shuffled ordering), plotted against elapsed time from the start of the leading idle window. The trace is a 2 s rolling median of samples taken at a nominal 100 ms interval (median 102 ms, measured from the record timestamps), shown at 1 s resolution. Benchmark segments are shaded and labelled and idle windows are shown in light grey, with boundaries taken from the UTC markers emitted by the harness. No thermally induced throttling is visible in any segment. Per-segment mean frequencies are given in Table 16.

No power or frequency telemetry was recorded on the Intel platform, so whether the halved issue rate lowers power there as it does on AMD is not established. The latency data nevertheless contain an indirect indication regarding frequency. The 512-bit kernels on Intel run uniformly slower than a cycle-exact halving predicts, in both the dependent and the independent kernel, while the frequency-independent quotient matches the documented cycle figures (Section 3.6). This is the pattern a small clock reduction during 512-bit execution would produce, and it is opposite in direction to AMD, where the 512-bit segments ran at a marginally higher measured frequency. We record it as an indication rather than a finding. Even taken at face value, it is more than an order of magnitude smaller than the issue-rate effect.

On AMD, the halved issue rate lowers power draw where lanes are fully occupied, so the time cost of the 512-bit path does not transfer proportionally into energy. The two effects act in opposite directions, and which dominates depends on lane occupancy and dependency structure (Section 3.10.1). This distinction is obscured by energy-delay reasoning applied to a single workload, and it is the principal methodological caution we would draw from these measurements.

For latency-bound sequential tasks such as postfix token evaluation, where a stack-carried dependency chain prevents concurrent issue, the additional issue capacity consumed by each 512-bit operation is not repaid by any per-element advantage. Targeting `Vector256` therefore yields the lowest measured latency on both platforms, by a clear margin on AMD and a marginal one on Intel. As the industry confronts the thermal limits of dark silicon, processor vendors are increasingly forced to balance parallel execution capabilities against strict power and heat constraints [21].

On AMD, where frequency was measured, the barrier to AVX-512 efficiency is issue capacity rather than frequency. The Intel latency data point the same way, although frequency was not measured there. In both cases software pipelines must therefore be context-aware rather than defaulting to the widest vector width. Part of the Vector256 advantage on AMD itself derives from AVX-512 encodings available to 256-bit code (Section 4.4), so the practical choice is a 256-bit width with the wider instruction set left enabled, not a return to AVX2. Surveys of the energy-efficiency mechanisms in Intel server processors document the costs attaching to frequency and voltage transitions, confirming that any theoretical gain from parallelization must outweigh the transition it triggers [22].

4.2. Neutralizing Garbage Collection for Race-to-Sleep Efficiency

In the context of Low Power Electronics, the concept of "race-to-sleep" (closely related to computational sprinting [23]) is critical. A processor minimizes its overall energy footprint by executing workloads as rapidly as possible and immediately transitioning back into deep C-states (idle states). In managed environments like .NET, this crucial energy-saving mechanism is frequently disrupted by GC. Direct power measurements of managed runtimes show that collection activity claims a disproportionate share of the total energy budget relative to its share of execution time [24]. Recent measurements on asymmetric multicore hardware show that where collection work is placed, and not only how much of it there is, measurably changes the energy consumed by an otherwise unchanged application. Moving concurrent collector threads onto energy cores yields roughly 3% savings at comparable latency [25].

The baseline .NET evaluation mechanism (`DataTableFull`) exemplifies this issue, allocating over 13 Kilobytes of transient objects per operation. Cross-language studies show that energy consumption tracks execution time far more closely than peak memory footprint, which makes the runtime cost of reclamation, rather than the allocated volume in itself, the operative variable [26]. A recent causal re-analysis strengthens this reading. Once implementation, active-core count and memory activity are controlled for, the choice of language implementation has no significant effect on energy beyond execution time [27]. The heap pressure of `DataTableFull` can be expected to trigger background collection cycles in both generations and so to extend the time the core spends in the active state beyond the arithmetic itself. Collection-induced wake-ups were not counted directly in this study. Measurements of thread management in managed runtimes describe the same pattern, with runtime services rather than application work dominating the energy envelope [28]. Section 3.10.2 quantifies the residency and energy consequence for the present pipeline.

By contrast, the proposed hybrid pipeline uses data structures that allocate no managed memory during the critical evaluation loop, where allocation is exactly 0 Bytes per operation. This property is specific to that stage, which receives pre-computed inputs. The complete pipeline allocates 1216 Bytes per operation against 13,368 Bytes for the `DataTableFull` control (Table 9). The collection rates follow the same ordering at the reference expression and widen under scaling, in both generations, as reported in Tables 7 and 8.

The duty-cycle experiment of Section 3.10.2 tests this mechanism directly rather than by inference, measuring the residency that race-to-sleep reasoning assumes but rarely measures, with the qualification stated there. This is consistent with systematic measurements showing that runtime configuration, and in particular the interaction of JIT compilation with the selected GC policy, measurably shifts the energy consumption of managed applications even when the application code is unchanged [29], and JIT compilation alone has been shown to make programs both faster and more energy-efficient in six of seven languages studied [30]. Where within that configuration the effect sits is narrowed by

a 2026 controlled study across three collectors, three workload intensities and two JDK distributions, in which workload intensity accounted for the dominant share of the energy variance, while collector selection on its own produced no statistically reliable effect [31]. What the runtime is asked to reclaim therefore matters more than which collector reclaims it, and it is the former that the present pipeline acts on.

4.3. Algorithmic Thresholds and Branch Predictability

The isolated benchmark results also revealed important algorithmic thresholds. For lexical analysis (`ParseOnly`), SIMD generally proved more efficient by mitigating branch mispredictions via bitmask extraction (`TrailingZeroCount`). This aligns with hardware-level studies showing that branch mispredictions not only waste execution cycles but also dissipate significant energy through speculative instruction fetching [32]. In the conversion stage, by contrast, the stack-free `ScanBasedConvert` strategy showed lower performance compared to the traditional scalar `ShuntingYardConvert` algorithm.

It was initially hypothesized that for sufficiently complex expressions the prefix-scanning overhead required to avoid explicit stack allocation would become advantageous once the working set exceeded L1 capacity. The empirical data does not support this hypothesis at any scale examined. At the reference length the difference is significant but of medium effect size, with a strongly right-skewed baseline distribution. Under scaling it widens decisively in favour of the stack-based algorithm, with `ShuntingYardConvert` at 29,270.54 ns against `ScanBasedConvert` at 38,556.84 ns at the 100× multiplier. The length sweep of Section 3.7 reinforces this. Where a crossover does appear at large input sizes, it runs against the flat-array approach rather than in its favour, and is driven by allocation rather than by cache locality.

The latency data establish that the stack-based algorithm outperforms flat-array scanning at every length examined, but do not by themselves identify the mechanism. A plausible explanation is that the Last-In-First-Out (LIFO) access pattern of an explicit stack keeps the active top of the stack in recently-touched L1 cache lines, whereas prefix-scanning algorithms sweep across wide, flat memory structures and evict active data as structural complexity grows. Confirming this would require cache-event counters that the present measurement does not provide. Work on in-cache computing is motivated by precisely this asymmetry, namely that the inefficiency of SIMD on non-contiguous data and the bandwidth of the cache itself, rather than the width of the vector registers, are what bound achievable parallelism [33].

Consequently, the localized cache coherence of a modern scalar stack proves resilient, suggesting that exploiting inherent stack locality remains an advantageous architectural strategy for expression parsing compared to flat-array scanning, even when memory footprints scale significantly. This is consistent with the classical result that reference locality and data layout, rather than raw instruction count, dominate the memory behaviour of pointer- and array-intensive programs [34]. From a microarchitectural perspective, a theoretical inflection point, where memory allocation overhead begins to outweigh these spatial locality benefits, would likely require extreme structural nesting depths capable of exceeding the L1 and L2 cache capacities. However, since real-world mathematical expressions typically exhibit relatively shallow nesting ratios compared to their overall length, the inherent cache-friendliness of this approach suggests that the Shunting Yard algorithm remains a highly effective and resilient strategy for standard parsing workloads.

4.4. Limitations and Future Directions

While this study implements three complementary measurement instruments on the AMD platform and reproduces the latency and allocation measurements on a second

microarchitecture, future research could further deepen this microarchitectural analysis. Deploying dedicated bare-metal profiling environments (e.g., custom Linux configurations with `perf` integration) would allow for the extraction of even more granular telemetry, such as direct ring-0 Intel Running Average Power Limit metrics and per-core voltage fluctuations, without OS-level virtualization interference. External hardware measurement would also complement the on-chip telemetry used here, since software-based estimates have been found to preserve the ranking obtained with a hardware power meter while absolute consumption and its per-component breakdown still require hardware measurement [35]. Additionally, extending the benchmark harness to include GPU-offloaded expression evaluation (e.g., via CUDA or DirectML) would provide a broader perspective on off-core energy efficiency for massively parallel mathematical tasks.

The workload coverage of this study remains partial. The present corpora vary input length across three orders of magnitude, nesting depth from one to fifteen in both associative directions, and rejection position across twelve malformed inputs, but they hold the operator distribution fixed and contain no unary operators or transcendental functions. Nor do they include randomly generated or application-derived expressions, which would test structural diversity beyond the controlled length and depth sweeps. Evaluating transcendental functions would shift the primary bottleneck from lexical analysis and memory management to floating-point execution throughput, and would likely move the workload toward the independent-operation regime in which, as Section 3.10 shows, the wider vector path becomes the more energy-efficient choice. Measurements on native 512-bit Intel silicon point the same way. AVX-512 implementations of two standard cryptographic primitives gain 1.16× and 1.61× over their AVX2 counterparts when processing a single message, and 2.28× once two messages are interleaved to expose instruction-level parallelism. The larger figure arises exactly where the dependency chain is broken, which is the regime our own throughput kernel isolates [36].

Five limitations bear directly on the claims made here. First, the halved 512-bit issue rate was observed through a single code generator on both platforms. Establishing that it is a property of the silicon rather than of RyuJIT requires reproducing it under a different toolchain, for which either CoreCLR on Linux or an equivalent kernel compiled from C with `<immintrin.h>` would suffice. Second, numerical agreement was established on the present corpora only (Section 3.8). Third, the power figures in Tables 16 and 17 are taken from the shuffled ordering. The throughput-kernel and pipeline comparisons hold in all three orderings, but in the evaluation regime neither the power ordering of the two vector widths nor the sign of their energy difference is consistent across orderings, and the Vector256Eval configuration exhibited 48.7% cross-ordering variance in rate. Fourth, the duty-cycle experiment establishes active-state residency and package energy but not C-state depth, which the power trace does not record. Relatedly, re-measuring with AVX-512 disabled (`DOTNET_EnableAVX512 = 0`, which restricts code generation to `x86-64-v3`) slowed the Vector256 evaluator by approximately 5%, well beyond its run-to-run variation. The scalar evaluator moved by a similar amount in the opposite direction, but within the variation observed between separate runs of the same configuration, so that effect is not established. Fifth, the present measurements characterize latency, power and allocation, but not microarchitectural events. Counter values obtained under `[HardwareCounters]` proved inconsistent with the effective core frequencies recorded during the same sessions. Non-determinism and systematic overcount in hardware counter implementations are documented across several x86 generations [37], and we read our own inconsistency as an instrumentation artefact of the same kind rather than as a property of the workload. These values are therefore excluded from the analysis, and the raw logs remain in the supplementary repository.

5. Conclusions

This study presents a comprehensive microarchitectural analysis of SIMD-accelerated mathematical expression parsing within MREs. By systematically isolating the lexical analysis, infix-to-postfix conversion, and numerical evaluation stages, we demonstrated that transitioning from traditional state-dependent algorithms to vectorized, branchless pipelines yields improvements in both computational latency and energy efficiency.

The empirical results indicate that allocation behaviour, rather than arithmetic throughput, dominates the practical characteristics of expression parsing in managed runtimes. The proposed architecture eliminates allocation entirely from the evaluation stage and reduces it by an order of magnitude relative to standard framework evaluators across the complete pipeline. Under a fixed duty cycle delivering identical work, this shortens active-state residency from 69.1% to 51.6% and reduces package energy by 15.9%, confirming the race-to-sleep benefit as a measured effect rather than an inferred one. The same analysis identifies where the approach reaches its limit. Beyond an input length of roughly thirty thousand characters, the per-invocation buffer required by the vectorized tokenizer promotes into the older generations and the advantage inverts.

Furthermore, the cross-architectural hardware profiling exposed a critical AVX-512 execution anomaly. The 512-bit path was slower in isolation than the 256-bit path on both platforms, by 5.7% on AMD Zen 4 and by 1.0% on Intel Tiger Lake. On both, 512-bit operations issue at half the rate of 256-bit ones, through double-pumping on Zen 4 and through a single port-fused unit on the client Tiger Lake core, so the wider register delivers no additional work per unit time. This aligns with independent measurements of hand-vectorized numerical kernels, in which AVX-512 code failed to outperform its AVX2 counterpart on silicon where the wider register width brings no proportional datapath advantage [38], providing empirical evidence that defaulting to the widest available SIMD register can inadvertently impact energy efficiency in certain contexts. This strongly supports the broader paradigm that less aggressive optimization, rather than maximal optimization, frequently yields superior time and energy behaviour [39].

Ultimately, achieving optimal performance in modern managed environments demands a hybrid, context-aware algorithmic approach. Three thresholds and one negative result emerged from this study and are, we suggest, more useful to practitioners than any aggregate speedup figure. Stack-based conversion outperformed flat-array scanning at every length examined, and we found no regime in which the scan-based strategy repaid its overhead. Vectorized tokenization is advantageous across three orders of magnitude of input length but inverts beyond approximately thirty thousand characters, where allocation cost dominates. The pipeline as a whole is the faster strategy below a few hundred evaluations of a given expression (approximately 230 on AMD and 310 on Intel), above which compiling once to a delegate is decisively superior. And the cost of the widest vector register on double-pumped silicon is expressed primarily in latency rather than in power where lanes are fully occupied, because the halved issue rate also lowers power draw, so that the wider path is then the more efficient choice. Matching vector width to lane occupancy and dependency structure, rather than to the widest available encoding, is the governing principle throughout.

Author Contributions: Conceptualization, P.K.; methodology, P.K.; software, P.K.; validation, A.I.; formal analysis, P.K. and A.I.; investigation, P.K.; resources, P.K. and A.I.; data curation, A.I. and P.K.; writing—original draft preparation, P.K.; writing—review and editing, P.K. and A.I.; visualization, A.I. and P.K.; supervision, A.I.; project administration, P.K.; funding acquisition, A.I. All authors have read and agreed to the published version of the manuscript.

Funding: This study is supported by the Centre of Excellence in Informatics and ICT under the Grant No BG16RFPR002-1.014-0018-C01, financed by the Research, Innovation and Digitalization for Smart Transformation Programme 2021–2027 and co-financed by the European Union.

Data Availability Statement: The complete experimental dataset is openly available in the supplementary Zenodo repository [17]. It comprises execution latencies and allocation logs for both evaluated microarchitectures, algorithmic scalability logs, and the power and frequency telemetry captured on the AMD platform. The source code of the parsing pipeline and of the benchmark and power-profiling harnesses is available from the corresponding author upon reasonable request.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AVX	Advanced Vector Extensions
DVFS	Dynamic Voltage and Frequency Scaling
EDP	Energy-Delay Product
ETW	Event Tracing for Windows
GC	Garbage Collection
JIT	Just-In-Time
MRE	Managed Runtime Environment
OS	Operating System
SIMD	Single Instruction, Multiple Data
TZCNT	Trailing Zero Count

References

1. Blackburn, S.M.; Cheng, P.; McKinley, K.S. Myths and realities: The performance impact of garbage collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*; ACM: New York, NY, USA, 2004; pp. 25–36. [CrossRef]
2. Cao, T.; Blackburn, S.M.; Gao, T.; McKinley, K.S. The yin and yang of power and performance for asymmetric hardware and managed software. In *Proceedings of the 2012 39th Annual International Symposium on Computer Architecture (ISCA)*; IEEE: New York, NY, USA, 2012; pp. 225–236. [CrossRef]
3. Nowatzki, T.; Menon, J.; Ho, C.H.; Sankaralingam, K. Architectural Simulators Considered Harmful. *IEEE Micro* **2015**, *35*, 4–12. [CrossRef]
4. Radhakrishnan, R.; Vijaykrishnan, N.; John, L.K.; Sivasubramaniam, A. Architectural issues in Java runtime systems. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture (HPCA)*; IEEE Computer Society: Washington, DC, USA, 2000; pp. 387–398. [CrossRef]
5. Gonzalez, R.; Horowitz, M. Energy dissipation in general purpose microprocessors. *IEEE J. Solid-State Circuits* **1996**, *31*, 1277–1284. [CrossRef]
6. Georges, A.; Buytaert, D.; Eeckhout, L. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA)*; ACM: New York, NY, USA, 2007; pp. 57–76. [CrossRef]
7. Mytkowicz, T.; Diwan, A.; Hauswirth, M.; Sweeney, P.F. Producing wrong data without doing anything obviously wrong! *ACM SIGPLAN Not.* **2009**, *44*, 265–276. [CrossRef]
8. Brooks, D.; Tiwari, V.; Martonosi, M. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*; ACM: New York, NY, USA, 2000; pp. 83–94. [CrossRef]
9. Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*; Order No. 248966; Intel Corporation: Santa Clara, CA, USA, 2024.
10. Aho, A.V.; Lam, M.S.; Sethi, R.; Ullman, J.D. *Compilers: Principles, Techniques, and Tools (2nd Edition)*; Addison-Wesley: Boston, MA, USA, 2006.
11. Polychroniou, O.; Raghavan, A.; Ross, K.A. Rethinking SIMD vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*; ACM: New York, NY, USA, 2015; pp. 1493–1508. [CrossRef]
12. Lemire, D. Scanning HTML at Tens of Gigabytes per Second on ARM Processors. *Softw. Pract. Exp.* **2025**, *55*, 1256–1265. [CrossRef]
13. Langdale, G.; Lemire, D. Parsing gigabytes of JSON per second. *VLDB J.* **2019**, *28*, 941–960. [CrossRef]

14. Lemire, D.; Boytsov, L. Decoding billions of integers per second through vectorization. *Softw. Pract. Exp.* **2015**, *45*, 1–29. [CrossRef]
15. Nizipli, Y.; Lemire, D. Parsing Millions of URLs per Second. *Softw. Pract. Exp.* **2024**, *54*, 744–758. [CrossRef]
16. Warren, H.S. *Hacker's Delight (2nd Edition)*; Addison-Wesley Professional: Boston, MA, USA, 2012.
17. Kyurkchiev, P. *Supplementary Dataset and Benchmark Logs: Energy-Efficient SIMD-Accelerated Expression Parsing*; Zenodo: Geneva, Switzerland, 2026. [CrossRef]
18. Laukemann, J.; Hammer, J.; Hofmann, J.; Hager, G.; Wellein, G. Automated Instruction Stream Throughput Prediction for Intel and AMD Microarchitectures. In *Proceedings of the 2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*; IEEE: New York, NY, USA, 2018; pp. 121–131. [CrossRef]
19. Advanced Micro Devices, Inc. *Software Optimization Guide for the AMD Zen4 Microarchitecture*; Publication No. 57647, Rev. 1.00; Advanced Micro Devices, Inc.: Santa Clara, CA, USA, 2023.
20. Bhargava, R.; Troester, K. AMD Next-Generation "Zen 4" Core and 4th Gen AMD EPYC Server CPUs. *IEEE Micro* **2024**, *44*, 8–17. [CrossRef]
21. Esmaeilzadeh, H.; Blem, E.; St. Amant, R.; Sankaralingam, K.; Burger, D. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*; ACM: New York, NY, USA, 2011; pp. 365–376. [CrossRef]
22. Hackenberg, D.; Schöne, R.; Ilsche, T.; Molka, D.; Schuchart, J.; Geyer, R. An Energy Efficiency Feature Survey of the Intel Haswell Processor. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*; IEEE: New York, NY, USA, 2015; pp. 896–904. [CrossRef]
23. Raghavan, A.; Luo, Y.; Chandawalla, A.; Papaefthymiou, M.; Pipe, K.P.; Wenisch, T.F.; Martin, M.M.K. Computational sprinting. In *Proceedings of the 2012 IEEE 18th International Symposium on High Performance Computer Architecture (HPCA)*; IEEE: New York, NY, USA, 2012; pp. 1–12. [CrossRef]
24. Hussein, A.; Payer, M.; Hosking, A.L.; Vick, C.A. Impact of GC design on power and performance for Android. In *Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR)*; ACM: New York, NY, USA, 2015; pp. 13:1–13:12. [CrossRef]
25. Shimchenko, M.; Österlund, E.; Wrigstad, T. Scheduling Garbage Collection for Energy Efficiency on Asymmetric Multicore Processors. *Art Sci. Eng. Program.* **2024**, *8*, 10. [CrossRef]
26. Pereira, R.; Couto, M.; Ribeiro, F.; Rua, R.; Cunha, J.; Fernandes, J.P.; Saraiva, J. Energy efficiency across programming languages: How do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*; ACM: New York, NY, USA, 2017; pp. 256–267. [CrossRef]
27. van Kempen, N.; Kwon, H.J.; Nguyen, D.T.; Berger, E.D. It's Not Easy Being Green: On the Energy Efficiency of Programming Languages. In Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE), Seoul, Republic of Korea, 16–20 November 2025; pp. 1553–1565. [CrossRef]
28. Pinto, G.; Castor, F.; Liu, Y.D. Understanding energy behaviors of thread management constructs. *ACM SIGPLAN Not.* **2014**, *49*, 345–360. [CrossRef]
29. Ournani, Z.; Belgaid, M.C.; Rouvoy, R.; Rust, P.; Penhoat, J. Evaluating the Impact of Java Virtual Machines on Energy Consumption. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*; ACM: New York, NY, USA, 2021; pp. 1–11. [CrossRef]
30. Maia, L.; Cunha, S.; Saraiva, J. Why Just-In-Time Compilation Matters: Evaluating Runtime and Energy Efficiency. In Proceedings of the 19th ACM SIGPLAN International Conference on Software Language Engineering (SLE), Rennes, France, 2–3 July 2026; pp. 92–104. [CrossRef]
31. Sharma, R. Garbage Collection and Energy Consumption in Java: A Controlled Study Across Workloads and JDKs. *arXiv* **2026**, arXiv:2608.19520. . [CrossRef]
32. Parikh, D.; Skadron, K.; Zhang, Y.; Barcella, M.; Stan, M.R. Power issues related to branch prediction. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*; IEEE: New York, NY, USA, 2002; pp. 233–244. [CrossRef]
33. Chen, Y.; Lee, M.; Dai, G.; Zhou, M.; Challapalle, N.; Wang, T.; Yu, Y.; Liu, Y.; Wang, Y.; Yang, H.; et al. GRAPHIC: Gather and Process Harmoniously in the Cache with High Parallelism and Flexibility. *IEEE Trans. Emerg. Top. Comput.* **2024**, *12*, 84–96. [CrossRef]
34. Chilimbi, T.M.; Hill, M.D.; Larus, J.R. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI)*; ACM: New York, NY, USA, 1999; pp. 1–12. [CrossRef]
35. Gordillo, A.; Calero, C.; Moraga, M.Á.; García, F.; Fernandes, J.P.; Abreu, R.; Saraiva, J. Programming Languages Ranking Based on Energy Measurements. *Softw. Qual. J.* **2024**, *32*, 1539–1580. [CrossRef]
36. Lee, S.W.; Song, M.H.; Kim, H.G.; Kim, U.J.; Eum, S.W.; Seo, H.J. Advanced Vector Extensions 512 Acceleration of LSH and LEA-GCM. *Appl. Sci.* **2026**, *16*, 6846. [CrossRef]
37. Weaver, V.M.; Terpstra, D.; Moore, S. Non-Determinism and Overcount on Modern Hardware Performance Counter Implementations. In Proceedings of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Austin, TX, USA, 21–23 April 2013; pp. 215–224. [CrossRef]

38. Watanabe, H.; Nakagawa, K.M. SIMD vectorization for the Lennard-Jones potential with AVX2 and AVX-512 instructions. *Comput. Phys. Commun.* **2019**, *237*, 1–7. [CrossRef]
39. Georgiou, K.; Blackmore, C.; Xavier-de Souza, S.; Eder, K. Less is more: Exploiting the standard compiler optimization levels for better performance and energy consumption. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems (SCOPES)*; ACM: New York, NY, USA, 2018; pp. 35–42. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.