Beyond Traditional Lexing

Exploiting SIMD Instructions for Tokenizing C

Alexandru Bolfa

Supervisor(s): Soham Chakraborty, Dennis Sprokholt

EEMCS, Delft University of Technology, The Netherlands

Abstract

Over the past decades, Single Instruction, Multiple Data (SIMD) instructions have become commonplace in conventional hardware. Lexical analysis, the first stage of compilation, can take advantage of this by splitting its workload across sub lexers that identify groups of tokens with similar structures. Each sub lexer can leverage SIMD to search for these structures across multiple characters in parallel and extract token positions efficiently. This paper presents a lexer with this architecture for the C programming language, along with implementation details for x86/x64 processors. Benchmark results show a 12x speed up in throughput compared to the lexer found in GCC, a state-of-the-art compiler.

1 Introduction

Compilers play a crucial role in software development, but prolonged compilation times can impact developer productivity due to frequent context switches. Lexical analysis/tokenization, commonly referred to as ”lexing”, is the first stage of compilation, converting raw text into a sequence of meaningful tokens that are more easily processed by subsequent stages. For example, the line of C code:

$$a \mathrel{+}= b$$

is broken down into identifiers a and b, with the addition assignment operator += in-between them.

Lexing time constitutes for approximately 10%[1] of compilation time in compilers like GCC [1] and should not be ignored. Optimizing this stage is desirable as faster compilers reduce waiting times and conserve computational resources, contributing to a more efficient software development processes.

Traditionally, lexers process source code character by character following a finite state machine. While some modern lexers employ SIMD operations to a limited extent, often for tasks like locating the end of comments, they generally handle one character at a time. While effective, this approach does not fully leverage modern hardware capabilities which can allow for parsing as many as 64 characters at once [2].

In this paper, we aim to answer the question: **”To what extent can SIMD instructions accelerate lexical analysis?”** To address this, we introduce a SIMD-oriented lexer architecture that revolves around the concept of sub-lexers, each responsible for identifying subsets of the original grammar. Our implementation specifically leverages SIMD instructions on the x86 architecture. We achieve quantifiable results by applying the proposed approach to the grammar of the C17 programming language [3]. Performance metrics, such as throughput, are measured during benchmarks and used for comparison with other designs.

[1]Based on empirical studies.

2 Background

Since 1996, SIMD instructions have been widely deployed to general purpose computers, starting with Intel’s MMX[2] [4] extensions to the x86 architecture [5]. They are designed to process multiple elements simultaneously using a single instruction. Consider the task of adding two vectors of integers:

$$\text{A} = [a_1, a_2, a_3, a_4]$$
$$\text{B} = [b_1, b_2, b_3, b_4]$$

Using SIMD instructions (e.g., SSE, AVX), you can add these vectors in parallel in one clock cycle:

$$\text{C} = [a_1 + b_1, a_2 + b_2, a_3 + b_3, a_4 + b_4]$$

Subsequent extensions beyond MMX, such as SSE (Streaming SIMD Extensions) [6], SSE2, SSE3, AVX (Advanced Vector Extensions) [7], and AVX-512, have continued to expand SIMD capabilities. These advancements have increased the width of SIMD registers, introduced more instructions, and enhanced performance across a broader range of applications.

Vectorization is the process of converting scalar operations, which process a single element at a time, into vector operations. Modern compilers attempt to automatically vectorize code to take advantage of SIMD. However, most SIMD instructions execute simple arithmetic and data movement operations that work best when elements act independently. As such, achieving optimal performance often requires writing code in a SIMD-friendly manner. This involves structuring data and devising algorithms to ensure that elements can be processed independently.

3 Related Work

Bernecky [8] notes that ”There appears to be a commonly-held feeling among the research community that non-numeric computations ... do not offer much parallelism.” to which he refers as ”An Embarrassment of Riches”. In his work, he proposes a parallel SIMD lexer for the A Programming Language (APL)[3], implemented in APL. Unlike C, which is scalar-centric, APL operates on arrays, making it well-suited for SIMD applications. Consequently, the APL lexer employs specific vector operations that do not translate directly to x86/x64 SIMD without a comprehensive understanding of APL.

Kartzke and Donegan [10] study a similar approach on the CDC STAR-100 [4] which boasts an ISA similar to that of x86/x64. However, the input source code that it parses is heavily restricted (according to modern standards). Input is assumed to consist either of comment or arithmetic statements and identifiers are no longer than 6 characters.

[2]Unofficially standing for MultiMedia eXtension.

[3]APL (named after the book A Programming Language) is a programming language developed in the 1960s by Kenneth E. Iverson [9].

[4]The CDC STAR-100 is a vector supercomputer from 1974 that was designed, manufactured, and marketed by Control Data Corporation (CDC) [11].

Steven R. House [12] has put forward the idea of splitting the lexer into multiple sub lexers for a mini programming language. The language provided only allows for integers and, to avoid ambiguities between identifiers and keywords, the latter are prefixed by the double-quotes.

Stepping outside the realm of compilers, JSON [13] parsing shares the same objective as lexing, that of organizing raw data into a structured format. State-of-the-art JSON parsers such as simdjson [14] rely on SIMD instructions to speed up text processing and gain four times performance over RapidJSON, ”the fastest traditional state-machine-based parser available” [15]. Although the JSON grammar is much simpler than that of C, it still shares common problems such as identifying escaped double quotes inside strings. Therefore, the impressive results achieved by simdjson on modern hardware suggests that leveraging SIMD in lexers can have similar results.

4 Lexer Scope and Responsibilities

The boundaries of a lexer’s responsibilities are not rigidly defined and can vary depending on the architecture of the compiler in which it is used. Before discussing any implementation details we need to highlight the extent to which raw text is going to be processed:

- Preprocessing-related grammar such as the syntax for directives like `#include`, `#define`, and conditional directives (`#ifdef`, `#ifndef`, `#if`, `#elif`, `#else`, `#endif`) will not be considered and are assumed to be handled beforehand by a preprocessor. The same assumption is made about digraph and trigraph replacements.
- Token contents are processed under the assumption of syntactical correctness, with validation and error handling deferred to subsequent stages. Consequently, a token like `1not_a_number2` would be recognized as a legal numeric constant[5].
- Input source code is limited to ASCII characters; Unicode and extended character sets are not within the lexer’s intended functionality.

Moreover, the lexer utilizes token definitions closely resembling those used by Clang.

5 Architecture and Implementation

One way to enable SIMD acceleration in lexical analysis is to split processing in multiple, smaller, vectorizable sub tasks whose results are then combined. This leads to the idea of sub lexers, smaller lexers which identify a subset of tokens that share structural similarities. Different sub lexers can find overlapping tokens, in which case we prioritize certain token types over others, given an ordering scheme.

We focus on AVX2 and earlier instructions which work on vectors of 32 characters at a time to speed up the lexing process. These instructions are widely supported on modern CPUs across different platforms, ensuring compatibility and future-proofing.

[5]Clang is an example of a modern compiler exhibiting similar behavior.

After loading the source code into memory, we process it in batches of 32 characters (a full vector), which are then passed to the sub-lexers. Each sub lexer generates a vector of tags that marks each character as either the beginning of a token or part of a token’s body.

The results from different sub lexers are merged using a prioritization scheme, and tokens are extracted from the tag vector by identifying the leading characters. This iterative process continues until all characters have been processed. Finally, an End of File (EOF) token is appended to signify the end of the token sequence. A diagram of this process can be seen in Figure 1.

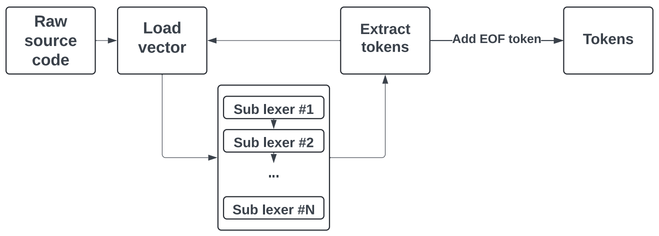

Figure 1: Architecture

We split the C17 grammar into nine SIMD-friendly lexical groups. These groups include:

- Four categories of **simple tokens** (which are always the same and do not require additional storage beyond their type):
  - Whitespaces
  - One-byte punctuators (+, `;`, ...)
  - Two-byte punctuators (+=, >>, ...)
  - Three-byte punctuators ((>>=, ...)
- Six categories of **complex tokens** (which contain additional data):
  - Identifiers (`foo`, ...)
  - Keywords (`int`, `auto`, ...)
  - Numbers (`2024`, `21.06`, ...)
  - Strings (`"foo"`, ...)
  - Characters (`’a’`, ...)
  - Comments (`// c`, `/* c */`, ...)

By their nature, complex tokens can span across multiple vectors which introduces horizontal data dependency between batches, something that SIMD struggles with. However, focusing solely on the leading characters of tokens allows us to interpret complex tokens like numbers, identifiers, and strings as implicit sequences of bytes with varying lengths. Figure 2 illustrates this concept.

```
int x += 1.E0;/* "c" */x>>= ’0’;
// Tags:
I__0I_200N___4_________i1000C004
```

Figure 2

This is referred as *in-situ*[6] and we do it by keeping pointers[7] to the start of tokens.

Since simple tokens are already defined by their types and do not need the in-string memory anymore after lexing. We can convert them to null-bytes to make complex-token null-terminated. This makes it easier for subsequent compiling stages to handle them, as in figure 3

```
// Before:
int x += 1.E0;/* "c" */x>>= ’0’;
// After:
int@x@@@@1.E0@@@@@@@@@@x@@@@’0’@
```

Figure 3: Input with simple tokens replaced by null-bytes (here represented by @).

For the remainder of this section we will describe in-depth all sub lexers and provide implementation details for the x86/x64 architecture.

5.1 Lookaround

SIMD code operates on vectors whose size depends on the CPU architecture. Consequently, tokens may span multiple vectors. However, for simplicity, the algorithms described in the following sections work as if data resides in a single large vector. This can be done because all sub lexers are designed to require minimal lookaround (*i.e.* lookahead and lookbehind), with the three-byte punctuator requiring the most (specifically, two bytes of lookahead). Using lookaround we abstract lexing from vector sizes.

5.2 White spaces

White spaces, such as spaces, tabs, and newline characters (line feed and carriage return), serve as delimiters between tokens but are not tokens themselves. To get a mask of these we do one comparison using `vpcmpeqb` (compare equal) for every type and the combine the results using `vpor` (logical OR).

After applying the mask, the vector tags is modified by replacing the masked positions with the white space tag. Figure 4 illustrates the updated vector.

```
int x += 1.E0;/* "c" */x>>= ’0’;
// Tags:
___0____0_______0___0______0____
```

Figure 4: The updated vector after the white spaces sub lexer, with 0 being the white space tag.

5.3 Punctuators

Punctuators are tokens whose syntactic and semantic meaning depends on the context [16]. For example, the asterisk (*) can represent both multiplication (`a * b`), pointer type definition (`int *x`), or dereferencing (`*ptr`).

[6]*In-situ* means processing or modifying data directly in its original location. In the context of tokenization, this approach involves not copying tokens to a separate data structure.

[7]We actually remember integer position in the original string as opposed to pointers to have a more compact memory layout.

Since some punctuators can contain other punctuators within them, we categorize punctuators by their length into one-byte, two-byte, and three-byte groups for lexing. To prevent conflicts, we prioritize the longest punctuators over the shorter ones.

Three Byte Punctuators

Three-byte punctuators are defined by the shift assign operators (<<=, >>=) and the ellipsis (`...`). They can be lexed by using vector comparisons to create four masks for each unique character and then do a combination of shifting and merging them to produce the a mask of the leading characters of each token.

To ensure that the remaining characters in the token do not get assigned to another token, we temporarily assign them the white space character as seen in figure 5.

```
int x += 1.E0;/* "c" */x>>= ’0’;
// Tags:
___0____0_______0___0___9000____
```

Figure 5: The updated vector after the three byte punctuator sub lexer.

Two Byte Punctuators

There are 19 two-byte punctuators (excluding preprocessing directives), including the arithmetic assignment operators (+=, -=, *=, /=, %=), the increment and decrement operators (++, --), the bitwise assignment operators (&=, |=, ˆ), the logical operators (&&, ||), the shift operators (<<, >>), the relational operators (<=, >=, ==, !=), and the arrow operator (->).

We identify them by applying the same logic as for the three-byte punctuators, getting a mask of the leading character of each token. We then keep only the leading characters whose positions have unassigned tags to not overwrite three-byte punctuators. Because writing to the tags vector has lower throughput than merging the masks, we found it faster to combine the masks of all two-byte punctuators and update the vector in one instruction using `vpblendvb` (blend packed bytes). Given a common mask we cannot differentiate between different token types, but by assigning each token the sum of the ASCII values of its characters minus two, we get tags that do not overlap with other token types.

Applying the described technique on the example input but with different tags for readability reasons is presented in figure 6.

```
int x += 1.E0;/* "c" */x>>= ’0’;
// Tags:
___0__200_______0___0___9000____
```

Figure 6: The updated vector after the two byte punctuator sub lexer.

One Byte Punctuators

There are 24 one-byte punctuators (excluding preprocessing directives), including arithmetic (+, -, *, /, %), comparison (<, >, =, !) and bitwise (&, |, ˆ, ˜) operators, as well as grouping symbols like parentheses ((,)), brackets

(`[, ]`), and braces (`{, }`). Additionally, they include separators such as the comma (`,`), semicolon (`;`), colon (`:`), period (`.`), and the ternary operator (`?`).

Although similar to white spaces in structure, due to the large number of characters, doing one comparison per item is very expensive. Lemire and Langdale [14] suggest a better approach called *vectorized classification* which finds a combined mask for a set of target characters. It utilizes the `vpshufb` (byte shuffle given lower nibble) instruction, which acts as lookup table, calling it twice to assign set memberships for the lower and upper nibbles of each character. Using `vpand` (Logical AND), we combine the two results, giving us a mask of one-byte punctuators in the input vector.

Before updating the tags vector, we remove from the mask periods that represent decimal points. Decimal points are periods that are either directly followed or which follow a numeric character. In syntactically correct C code, these periods are always part of numeric constants and are not tokens on their own and as such, we remove them from the mask. This can be done by creating a mask of numeric characters and one for periods. Using `vpor` (Logical OR) on a left shifted numeric mask and a right shifted one, we get a mask of all positions which have at least one numeric character as a neighbour. Combining this mask with the periods one gives us the decimal points that we then remove from the one-byte punctuators mask using `vpxor` (logical exclusive OR). In the case of illegal C code, we expect subsequent stages to catch the error.

Finally, using the mask created by *vectorized classification*, the tags vector can updated by assigning to each punctuator a token tag equivalent with the ASCII value of its character. In figure 7 we assign a different value than the ASCII one for improved readability.

```
int x += 1.E0;/* "c" */x>>= '0';
// Tags:
___0__200____45607_7065_10008_84
```

Figure 7: The updated vector after the one byte punctuator sub lexer.

5.4 Identifiers

Identifiers serve as names for variables, functions, constants, and types, and they consist of letters, digits, and underscores. Because identifiers have varied-length we lex them by marking only the starting character which in this case can be either a letter or an underscore. A character is starting if it follows a white space or punctuator token.

A mask of starting characters is get by merging a shifted right mask of white spaces or punctuators with a mask of English letters or underscores. The first mask is get by combining with `vpor` the white spaces mask with the punctuators mask from before. The second mask we can get through range searches in the input vector, specifically the range of lower case (97 - 122) and upper case (65 - 90) English alphabet letters.

Efficient range searches can be done using `pcmpistrm` instruction (compare implicit length strings return mask) which takes a list of ranges and returns a mask of all bytes falling in at least one of those ranges.

The updated tags vector is portrayed in figure 8.

```
int x += 1.E0;/* "c" */x>>= '0';
// Tags:
I__0I_200____45607I7065I1000_004
```

Figure 8: The updated vector after the one byte punctuator sub lexer.

Keywords

Keywords are identified by comparing tokens against a predefined list of reserved words specific to the programming language. Initially, all tokens are treated as identifiers. Each token is subsequently examined against the list of reserved words to check if it corresponds to any keywords; if a match is found, it is designated as a keyword; otherwise, it retains its status as an identifier.

For efficient token matching against the reserved words list, Lemire [17] introduces a method employing a non-collision hash function. This function maps the initial bytes of each identifier to a 1-byte key. The reserved keywords are stored in a lookup table indexed by these keys. Upon processing each identifier, its hash key is computed, and the corresponding entry in the lookup table is checked to determine if it matches the identifier, ensuring equality using `vpcmpeqb`.

5.5 Numbers

In the C programming language, numeric constants fall in two categories: integers and floating point numbers. Integers are represented as continuous sequence of digits. Floating point numbers can be written in decimal notation which is described by an optional[8] sequence of digits followed by another sequence of digits with a decimal point in between. Additionally, floating point numbers can accept different exponent notations such as scientific notation (*e.g.* 1E2).

Moreover, numbers can have trailing characters defining their type (*e.g.* 1u being unsigned integer). In order to cover the vast range of numeric constants and also allow for compiler extensions, we consider numbers all sequences starting with a digit or period that are followed by alphanumeric characters or other periods[9]. However, since we only mark the leading character, it is enough to create a mask of digits and all *unassigned* periods (which are decimal points) and keep only the ones that follow directly either a white space or a punctuator. We do this by using the previous punctuators and white space masks, that we shift by one to the right and merge with the digits-period mask using `vpand`.

Getting a digits mask can be done by range searching for characters in digits range (from 48 until 57 in ASCII). For range searches we use `pcmpistrm`.

By this stage, a representation of the tags vector can be seen in figure 9, where we mark number tags with the letter `n`.

5.6 Strings and Character constants

Strings are sequences of (escaped) characters surrounded by double quotes, while character constants represent one (escaped) character surrounded by single quotes. In the context

[8] `.1` is a legal floating point number with the value 0.1.

[9] Similar behaviour can be seen in Clang.

```
int x += 1.E0;/* "c" */x>>= '0';
// Tags:
I__0I_200N___45607I7065I10000N04
```

Figure 9: The updated vector after the numbers sub lexer.

of lexical analysis, character constants are not validated and their length is not checked (for example 'ab' will be a legal token). For all intents and purposes, we treat characters as strings with a different enclosing character. For the rest of this section we explain string lexing but the same logic is applied for character constants.

Strings represent the most complex token type because they can hold inside of them other tokens that should not be extracted. Moreover, strings can contain escaped ending characters which need to be detected beforehand. Lemire and Langdale [14] proposed a fully branchless SIMD algorithm for solving this problem. It first finds a mask of all double quote characters by vector comparison. Then it finds a mask of escaped characters using a formula[10] that uses the mask of even-indexed positions, odd-indexed positions and a mask of backslashes. The second mask is used to remove escaped double quotes from the first mask.

Lemire and Langdale [14] then show that to get a mask of the strings, we can do prefix XOR on the unescaped double quotes mask. Prefix XOR can be done in a SIMD manner by doing `pclmulqdq` (carry-less multiplication) between the mask and a vector full of 1's.

If a string is longer than the vector size, then we assign the first character of the next vector to a string so the string continues. Similarly to identifiers and numbers, we assign string tags to double quotes that follow white space character and which are not part of the body of any strings.

In the end, we take the mask of strings and use it to mark the start of the string with a tag but also to remove any tokens found inside the string. To make sure that the body of the string does not get assigned to another token, we temporarily mark it with the white space tag as seen in figure 10.

We do the same process for character constants as well.

```
int x += 1.E0;/* "c" */x>>= '0';
// Tags:
i__0i_200n___4560S00065i1000C004
```

Figure 10: The updated vector after the string and character literals sub lexers.

5.7 Comments

Comments represent text that is ignored by the compiler. Usually they are used for improving code readability by explaining and clarifying code. Similar to strings, comments may include tokens that need to be discarded but unlike strings, we do not store comments as different tokens.

There are two type of comments: line comments (starting with `//`, ending with *newline*) and block comments (starting with `/*`, ending with `*/`). For both types of comments, we create a mask of the starting and ending delimiters which are not inside strings, and then apply prefix XOR on it. If the comments do not contain starting delimiters in them, then a mask of each comment has been created. If the comments contain starting delimiters in them, then the mask considers ill-formed comments that end at a starting delimiter. To fix this, we remove the `*/` that comes after another `*/` and re-run the process. That is, we remove all starting delimiters whose position is 0 in the prefix XOR. After at most three runs, we get a mask of the comments, however most code comments do not include other comments inside and we expect that on average this does not need any rerunning.

In a similar fashion, for line comments newlines that start comments need to be removed. That is, newlines whose prefix XOR value is 1 (one). A mask of all comments is then formed by merging the two masks and it is used to remove all tags found inside it as shown in figure 11

If a comment is longer than the vector size, then we assign the first character of the next vector to a comment so the comment continues.

```
int x += 1.E0;/* "c" */x>>= '0';
// Tags:
I__0I_200N___4_________i1000S004
```

Figure 11: The updated vector after the comments sub lexer.

5.8 Token extraction

After all sub lexers finish, there is a vector of tags marking the start of all tokens. We use it to extract token values and then store them in an Structure of Arrays (SoA)[11].

Using a vector of positions and `pext` (parallel bits extract) we can extract all tokens and their positions in a SIMD manner like in figure 12 and 13. The extracted values can then be stored efficiently in memory using `vmovdqu` (move unaligned packed integer values).

```
int x +=
// Tags:
I___i_2_
// Positions:
01234567
```

Figure 12: First 8 characters.

After the last batch of tokens was extracted, the EOF (end of file) token is appended, finalising the lexing phase.

6 Experiments

Experiments are conducted to evaluate the performance of the proposed architecture and analyze its behaviour dealing with

[10]That is: `(((B + (B &~(B << 1)& E))& ~B)& ~E) | (((B + ((B &~(B << 1))& O))& ~B)& E)`. Here, `B` is the mask of backslashes, `E` is the mask of even-indexed positions and `O` is the mask of odd-indexed positions [14].

[11]The Structure of Arrays (SoA) layout stores data in separate arrays which allows for SIMD loading and processing.

```
// Tags:
Ii2______
// Positions ('d' means "do not care"):
046ddddd
```

Figure 13: First 8 characters after `pext`.

different types of input. Lastly, they allow for a better understanding of its role compared to alternative implementations.

Input source code is loaded into memory and pre processed (as outlined in Section 4) before running any experiments. By isolating I/O operations, we obtain more accurate and reliable benchmark results, focusing only on the lexer's processing efficiency. Additionally, the preprocessing stage in essential to avoid feeding unknown characters to the lexer (such as # used in the context of directives). For the purpose of this paper, we use GCC's preprocessor and its output as printed by the flags `-E -C`[12] from which we remove preprocesser's comments that start with #, using `grep -v ^#`.

The effectiveness of SIMD instructions relies heavily on the specific hardware they are executed on. For these experiments we will run the lexers on an AMD processors with the specifications in table 3.

Table 1: Hardware

Processor	Microarchitecture	Base Hz	Max Hz
AMD Ryzen 9 5900HS	Zen 3	3 GHz	4.6 GHz

The lexer is written in C11 (CLANG 18) using SIMD intrinsics. The code is compiled with the *-O3* flag for maximum performance. We use the AVX2 instruction set and 256-bit vector registers and we call SIMD instructions using intrinsics. In each benchmark we time using `clock` the period of time spent lexing.

To have a better understanding of the results, we compare them to existing solutions. For this we picked a collection of three lexers covering the most prominent active implementations. CHIBICC [18] is a toy compiler with a traditional lexer implementation, following a finite state machine. Simd-lexer (to which we will refer to as Mateuszd6/simd-lexing) written by Mateusz Dudziński (with GitHub tag: Mateuszd6) [19] represents an alternative SIMD implementation that uses SIMD for lookahead. GCC is a state-of-the art compiler whose lexer can be accessed through various command line flags. Their exact version is referenced in table 2. From now on, we will refer to the implementation of the method proposed in Section 5 as **simd-lex**.

6.1 Throughput

Throughput in the context of lexing refers to the rate at which a lexer processes input data. Calculating throughput is crucial as it provides insights into the efficiency and performance of the lexer implementation.

Throughput of different designs is closely linked with the input provided as some approaches deal better with certain peculiarities of input (such as heavily commented code or very short variable names). For this reason, we benchmark results of the lexers outlined in table 2 against a set of publicly available C single compilation units listed in 3.

[12]The `-E` flag instructs GCC to stop after the preprocessing stage, and the `-C` flag retains comments in the preprocessed output.

Table 2: Comparison lexers

Lexers used for comparison.

Lexer	Snapshot
https://github.com/Mateuszd6/simd-lexing	June 23, 2024
https://github.com/rui314/chibicc	June 23, 2024
GCC	13.1

Table 3: Files

Single compilation unit files used for benchmarking and the number of lines of code (LoC) after the preprocessing stage.

File	Preprocessed LoC
GCC	734k
oggenc	73k
gzip	21k
people.csail.mit.edu/smcc/projects/single-file-programs	

CHIBICC and Mateuszd6/simd-lexing and simd-lex are timed by recording the clock time at the start of lexing and at the end. Due to it's complex nature, timing GCC is more complicated. However, using the flags `-ftime-report -fsyntax-only` we get a report of how much time it was spent in each stage of the compiler's frontend. We will use "lexical analysis" wall time as measurement for the speed of GCC's lexer. These experiments were run 100 times and on the device covered in table 3 and their results can be seen in figure 14.

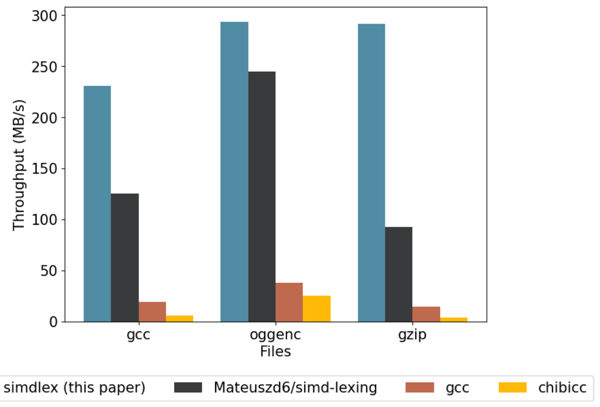

Figure 14: Throughput on AMD Zen 3 (higher is better).

Moreover, input size can have an impact into algorithmic performance. To check for this, we determine the throughput of different synthetically created source code of varying sizes. Figure 15 plots such data. From it we observe a speed increase as input grows in the beginning, but converges by the end. The initial growth can be attributed to ramp up[13].

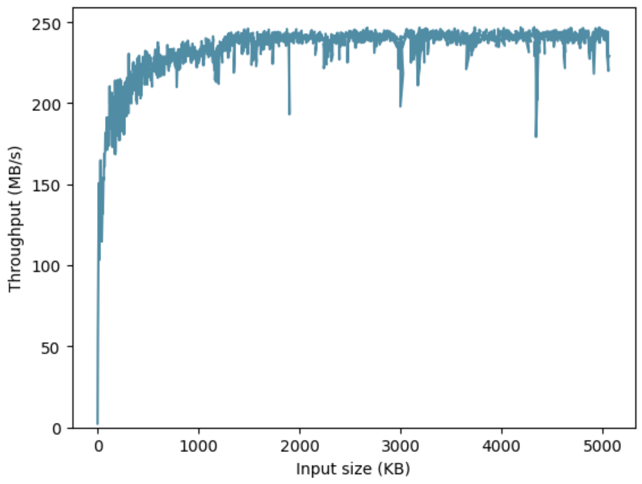

Figure 15: Throughput on synthetic data (higher is better).

7 Responsible Research

7.1 Proof of correctness

C programs are prone to bugs due to the language's low-level operations and lack of built-in safety features. This vulnerability is particularly concerning given the widespread use of C in embedded software, which has led to several catastrophic bugs. For instance, in 2009, a stack overflow in the software of a Toyota car caused unintended acceleration, resulting in the deaths of four people [20] [21].

Compilers are crucial in ensuring the correctness of code logic. A flawed compiler can misinterpret correct code, leading to unintended behavior. In this context, lexical analysis is vital to ensure that tokens are correctly identified and processed.

Formal proofs are ideal for validating the techniques discussed in this paper, however, time constraints led to the use of empirical testing to assess correctness. This means that it is up to engineers to consider worst-case scenarios and make responsible decisions on whether the proposed ideas fits their project.

7.2 Reproducibility

To ensure the transparency and reliability of our research findings, all tests conducted throughout this study were automated[14], allowing for consistent and repeatable results. Furthermore, an implementation of the proposed technique has been made publicly available on GitHub[15], accessible to all interested parties for review and replication.

8 Conclusion and Future Work

In this paper we aimed to see **the extent to which SIMD instructions found in conventional x86/x64 hardware can improve lexical analysis**. Given the results from our experiment we draw the conclusion that significant speeds ups can be seen over mainstream lexers such as that employed by GCC (by a 12x speed up). While results are promising, it is important to keep in mind that the proposed architecture is more complex than what is currently used by other lexers.

Similar SIMD-oriented lexer architectures can be developed for other programming languages, particularly those with grammars similar to that of C, such as Java and C++. We expect achieving comparable improvements in these contexts.

To get a better understanding of the lexer's performance it is important to test it across different processor architectures and micro-architectures[16]. This should provide generalizability of our findings and even boost performance with the advent of AVX512, which introduces larger registers and new instructions.

Looking ahead, future efforts could focus on parallelizing lexing using Multiple Instruction Multiple Data (MIMD) techniques, such as multithreading. This approach could potentially enhance performance by executing multiple sub-lexers in parallel, with results aggregated at the token extraction stage. This should be possible as multiple sub-lexers run independently of each other.

Additionally, extending these optimizations to the ARM architecture family presents a compelling avenue for future research, in which we expect to achieve similar performance gains to those observed in x86/x64 architectures.

8.1 Acknowledgements

We would like to express our gratitude to Daniel Lemire and Geoff Langdale for their significant contributions in advancing SIMD capabilities in test processing. Additionally, we extend our thanks to Arav Khanna for his valuable insights in using SIMD.

References

[1] Free Software Foundation, *GNU Compiler Collection (GCC) Manual*. Free Software Foundation, 2024.

[2] I. Corporation, "Intel® Advanced Vector Extensions 512 (Intel® AVX-512)," *Intel Corporation*, 2017.

[3] "Programming languages – C," 2018. ISO/IEC 9899:2018.

[4] I. Corporation, "Intel architecture mmx technology," 1996. Accessed: 2024-06-20.

[5] Wikipedia contributors, "Single instruction, multiple data — Wikipedia, the free encyclopedia." https://en.wikipedia.org/wiki/Single_instruction,_multiple_data, 2024. Accessed: May 8, 2024.

[6] I. Corporation, "Intel architecture sse technology," 1999. Accessed: 2024-06-20.

[7] I. Corporation, "Intel advanced vector extensions (intel avx)," 2011. Accessed: 2024-06-20.

[13] Ramp up is the process of filling up the CPU's pipeline.

[14] https://github.com/alexbolfa/simd-lex-experiments

[15] https://github.com/alexbolfa/simd-lex

[16] Microarchitecture refers to the specific design or implementation of a CPU at the internal level.

[8] R. Bernecky, “An spmd/simd parallel tokenizer for apl,” in *Proceedings of the 2003 Conference on APL: Stretching the Mind*, APL ’03, (New York, NY, USA), p. 21–32, Association for Computing Machinery, 2003.

[9] Wikipedia contributors, “APL (programming language).” https://en.wikipedia.org/wiki/APL_(programming_language), 2024. [Online; accessed 3-June-2024].

[10] M. K. Donegan and S. W. Kartzke, “Lexical analysis and parsing techniques for a vector machine,” *SIGPLAN Not.*, vol. 10, p. 138–145, jan 1975.

[11] W. contributors, “Cdc star-100.” https://en.wikipedia.org/wiki/CDC_STAR-100, [year when the page was last updated, or n.d. if not available]. Accessed [Access Date].

[12] S. R. House, “Compiling in parallel,” in *Conpar 81* (W. Brauer, P. B. Hansen, D. Gries, C. Moler, G. Seegmüller, J. Stoer, N. Wirth, and W. Händler, eds.), (Berlin, Heidelberg), pp. 298–313, Springer Berlin Heidelberg, 1981.

[13] T. Bray, “The JavaScript Object Notation (JSON) Data Interchange Format.” RFC 8259, Dec. 2017.

[14] G. Langdale and D. Lemire, “Parsing gigabytes of json per second,” *The VLDB Journal*, vol. 28, p. 941–960, Oct. 2019.

[15] S. Palkar, F. Abuzaid, P. Bailis, and M. Zaharia, “Filter before you parse: faster analytics on raw data with sparser,” *Proceedings of the VLDB Endowment*, vol. 11, pp. 1576–1589, 07 2018.

[16] IBM, “Ibm xl c/c++ for aix documentation: Tokens, punctuators, and operators,” 2022.

[17] D. Lemire, “Recognizing string prefixes with simd instructions,” 2023. Accessed: 2024-06-12.

[18] rui314, “Chibicc: A Small C Compiler,” 2024. Accessed: June 16, 2024.

[19] Mateuszd6, “simd-lexing: SIMD Lexer Implementation,” 2024. Accessed: June 16, 2024.

[20] S. A. Bowen and Y. Zheng, “Auto recall crisis, framing, and ethical response: Toyota’s missteps,” *Public Relations Review*, vol. 41, no. 1, pp. 40–49, 2015.

[21] Wikipedia contributors, “2009–2011 toyota vehicle recalls.” https://en.wikipedia.org/wiki/2009%E2%80%932011_Toyota_vehicle_recalls, 2023. Accessed: 2024-05-24.