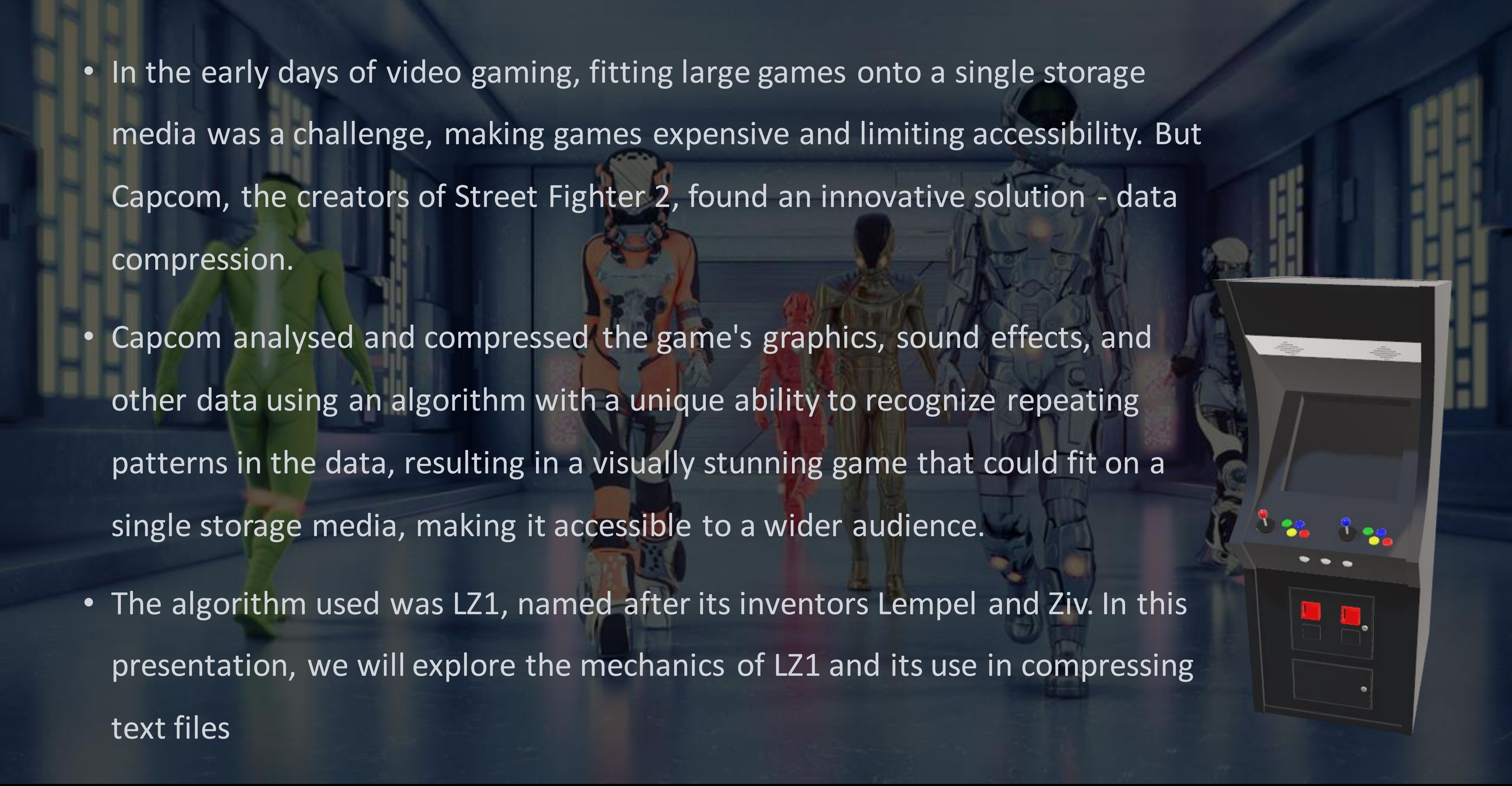
LZ1 Sequential Implementation

Presenters:
Rahul Mittal
Neel Prajapati

CAFE ELIZA
Subway
DOGS
CAPCOM

GAME OVER
GAME OVER

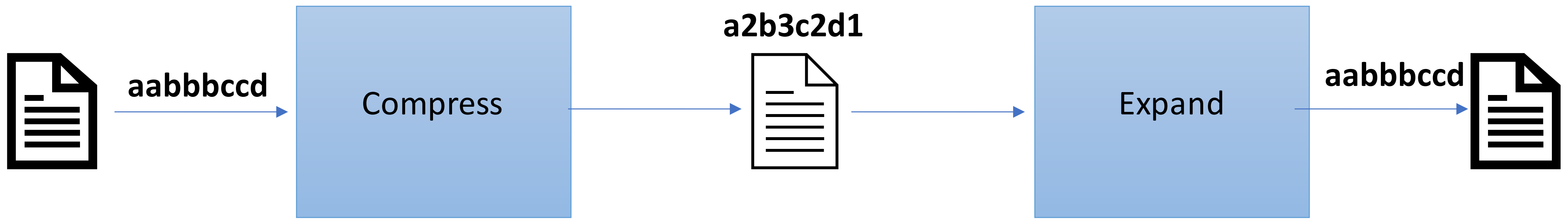


- In the early days of video gaming, fitting large games onto a single storage media was a challenge, making games expensive and limiting accessibility. But Capcom, the creators of Street Fighter 2, found an innovative solution - data compression.
- Capcom analysed and compressed the game's graphics, sound effects, and other data using an algorithm with a unique ability to recognize repeating patterns in the data, resulting in a visually stunning game that could fit on a single storage media, making it accessible to a wider audience.
- The algorithm used was LZ1, named after its inventors Lempel and Ziv. In this presentation, we will explore the mechanics of LZ1 and its use in compressing text files

Data Compression?

- Data compression is the process of reducing the size of a computer file. Through an algorithm or a set of rules for carrying out an operation, computers can determine ways to shorten long strings of data and later reassemble them in a recognizable form upon retrieval.

Basic Model for Data Compression



Data Has Repetitions!

- Data compression algorithms save space by exploiting the fact that most data files have many repeating patterns in them.
- For example, text data has certain character sequences that appear much more often than others

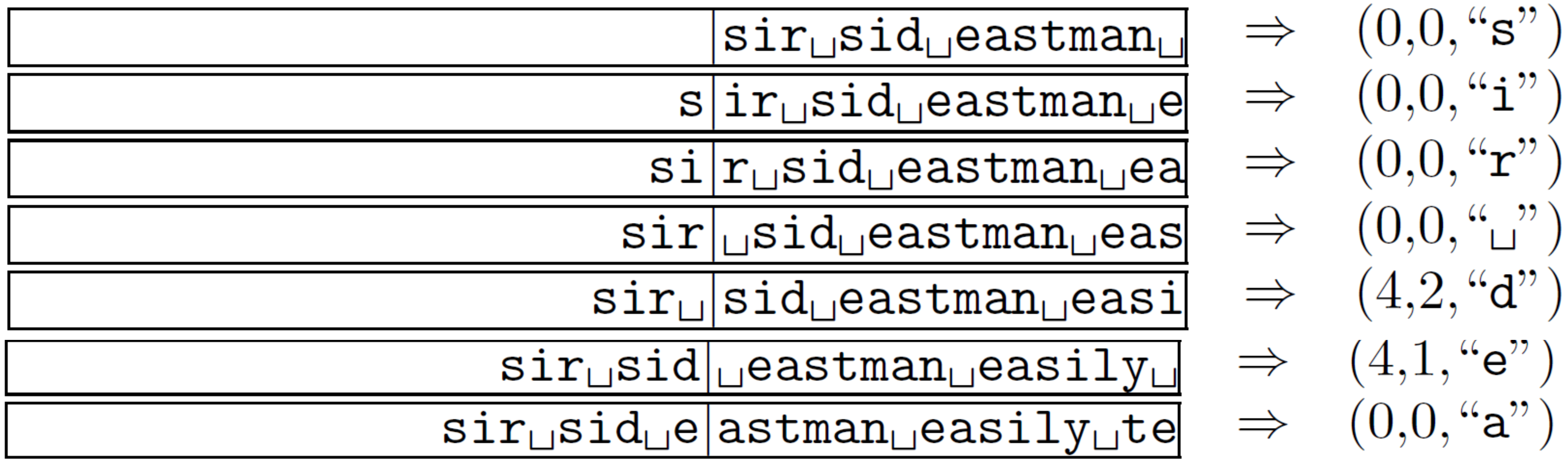
Repetitions Are Beneficial

- Identifying repeating patterns makes it possible to represent them using less space.
- One approch to identification and space saving is to point to past occurrence of the pattern.
- LZ1 algorithm, that is the topic of this presentation, is one of the members of a well know family of algorithms that uses adaptive dictionaries as the reference mechanism to point to past occurrences.

LZ1 Algorithm

```
while (input is not empty)
    {
        get a reference (position, length) to longest match;
        if (length > 0)
        {
            output (position, length, next symbol);
            shift the window length+1 positions along;
        }else {
            output (0, 0, first symbol in the lookahead buffer);
            shift the window 1 character along;
        }
    }
```

Compression Example



	sir␣sid␣eastman␣	⇒	(0,0,“s”)
s	ir␣sid␣eastman␣e	⇒	(0,0,“i”)
si	r␣sid␣eastman␣ea	⇒	(0,0,“r”)
sir	␣sid␣eastman␣eas	⇒	(0,0,“␣”)
sir␣	sid␣eastman␣easi	⇒	(4,2,“d”)
sir␣sid	␣eastman␣easily␣	⇒	(4,1,“e”)
sir␣sid␣e	astman␣easily␣te	⇒	(0,0,“a”)

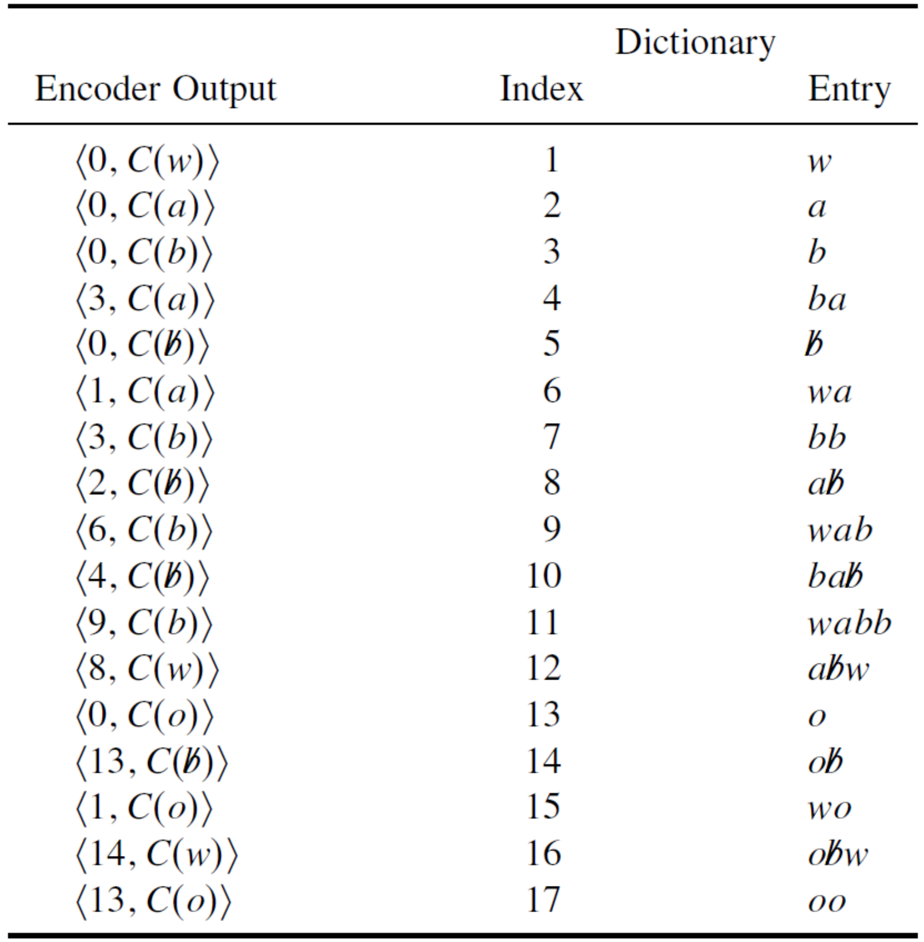
Source: Data Compression-The Complete Reference

An Advantage of Sliding Window

As the search window contains characters that are directly preceding the current characters being compressed, patterns repeating in only certain parts of the input are captured efficiently and effectively.

wabbaƀwabbaƀwabbaƀwabbaƀwooƀwooƀwoo

(Source: Introduction to Data Compression)

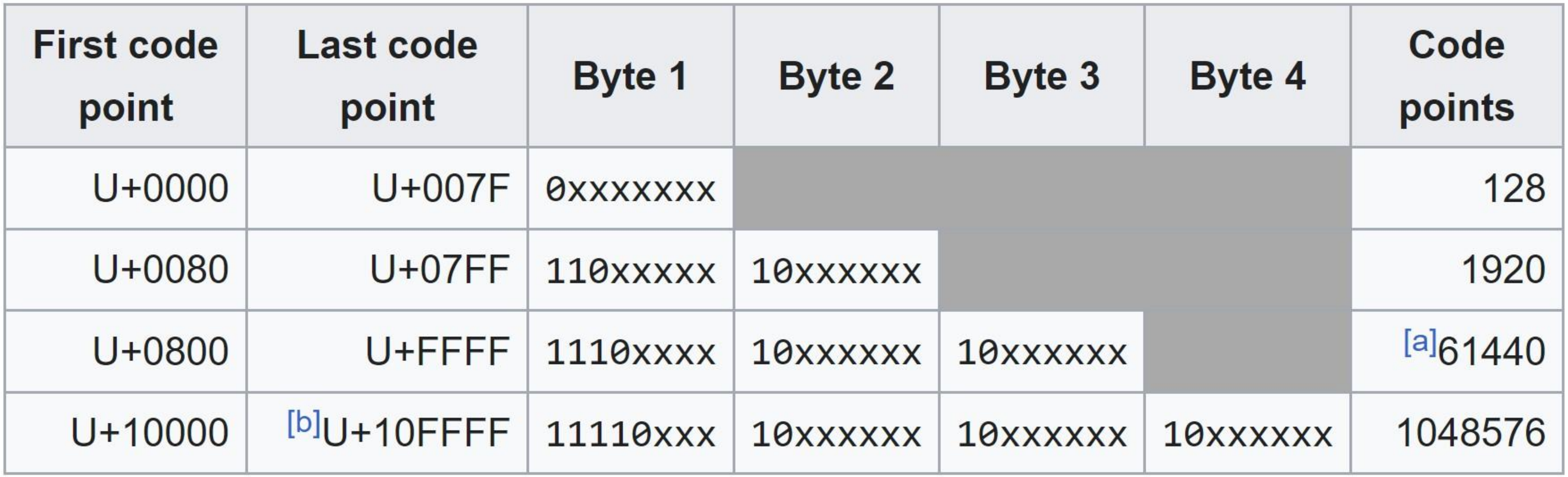


Encoder Output	Dictionary Index	Entry
$\langle 0, C(w)\rangle$	1	*w*
$\langle 0, C(a)\rangle$	2	*a*
$\langle 0, C(b)\rangle$	3	*b*
$\langle 3, C(a)\rangle$	4	*ba*
$\langle 0, C(\text{ƀ})\rangle$	5	*ƀ*
$\langle 1, C(a)\rangle$	6	*wa*
$\langle 3, C(b)\rangle$	7	*bb*
$\langle 2, C(\text{ƀ})\rangle$	8	*aƀ*
$\langle 6, C(b)\rangle$	9	*wab*
$\langle 4, C(\text{ƀ})\rangle$	10	*baƀ*
$\langle 9, C(b)\rangle$	11	*wabb*
$\langle 8, C(w)\rangle$	12	*aƀw*
$\langle 0, C(o)\rangle$	13	*o*
$\langle 13, C(\text{ƀ})\rangle$	14	*oƀ*
$\langle 1, C(o)\rangle$	15	*wo*
$\langle 14, C(w)\rangle$	16	*oƀw*
$\langle 13, C(o)\rangle$	17	*oo*

UTF-8 Representation

SPACE SAVER BECAME A NIGHTMARE

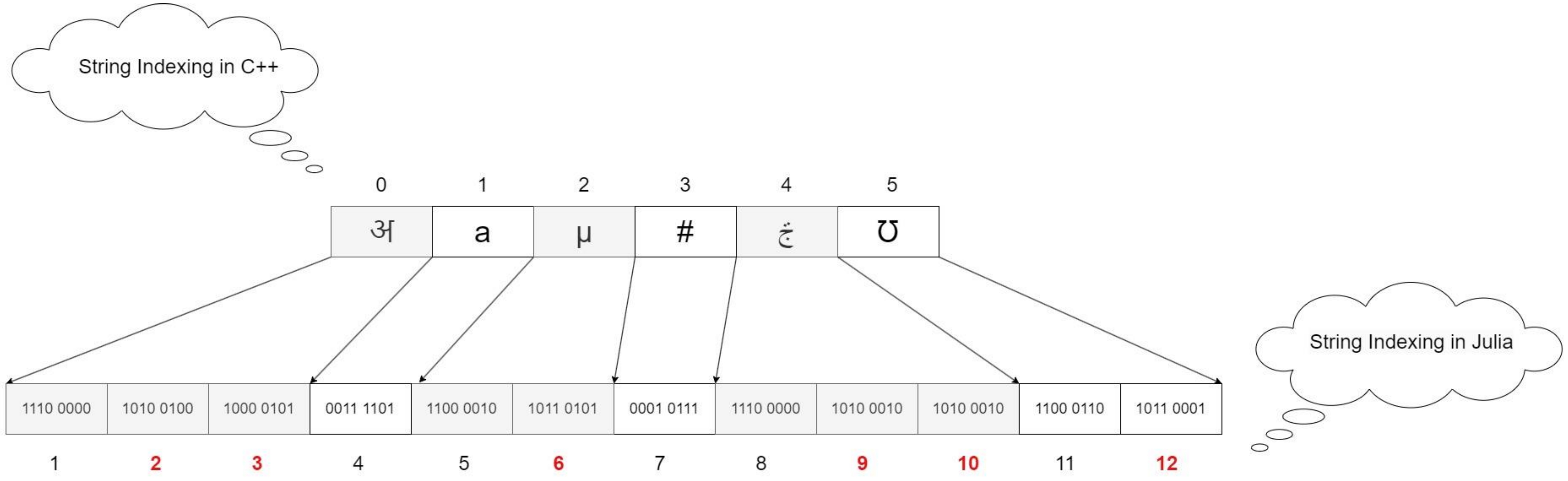
Code point ↔ UTF-8 conversion

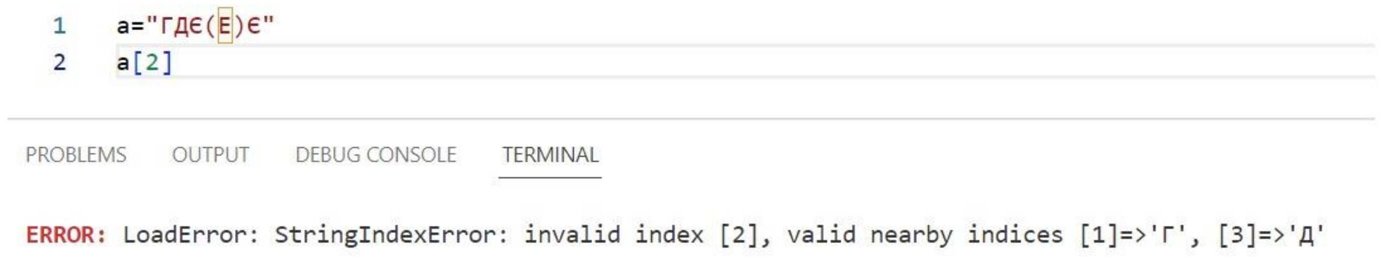


First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4	Code points
U+0000	U+007F	0xxxxxxx				128
U+0080	U+07FF	110xxxxx	10xxxxxx			1920
U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx		[a]61440
U+10000	[b]U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx	1048576

Source: Wikipedia

Challenges Faced





```
a="ГДЄ(Ё)Є"
a[2]
```

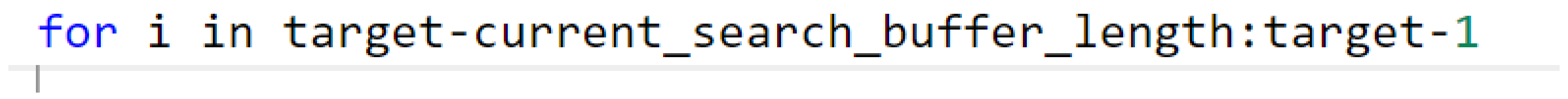
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
ERROR: LoadError: StringIndexError: invalid index [2], valid nearby indices [1]=>'Г', [3]=>'Д'
```

Julia String Indices are mapped to bytes.

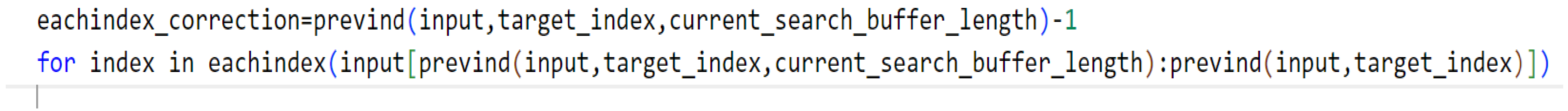
CODE COMPARISON

Restricted to ASCII in UTF-8



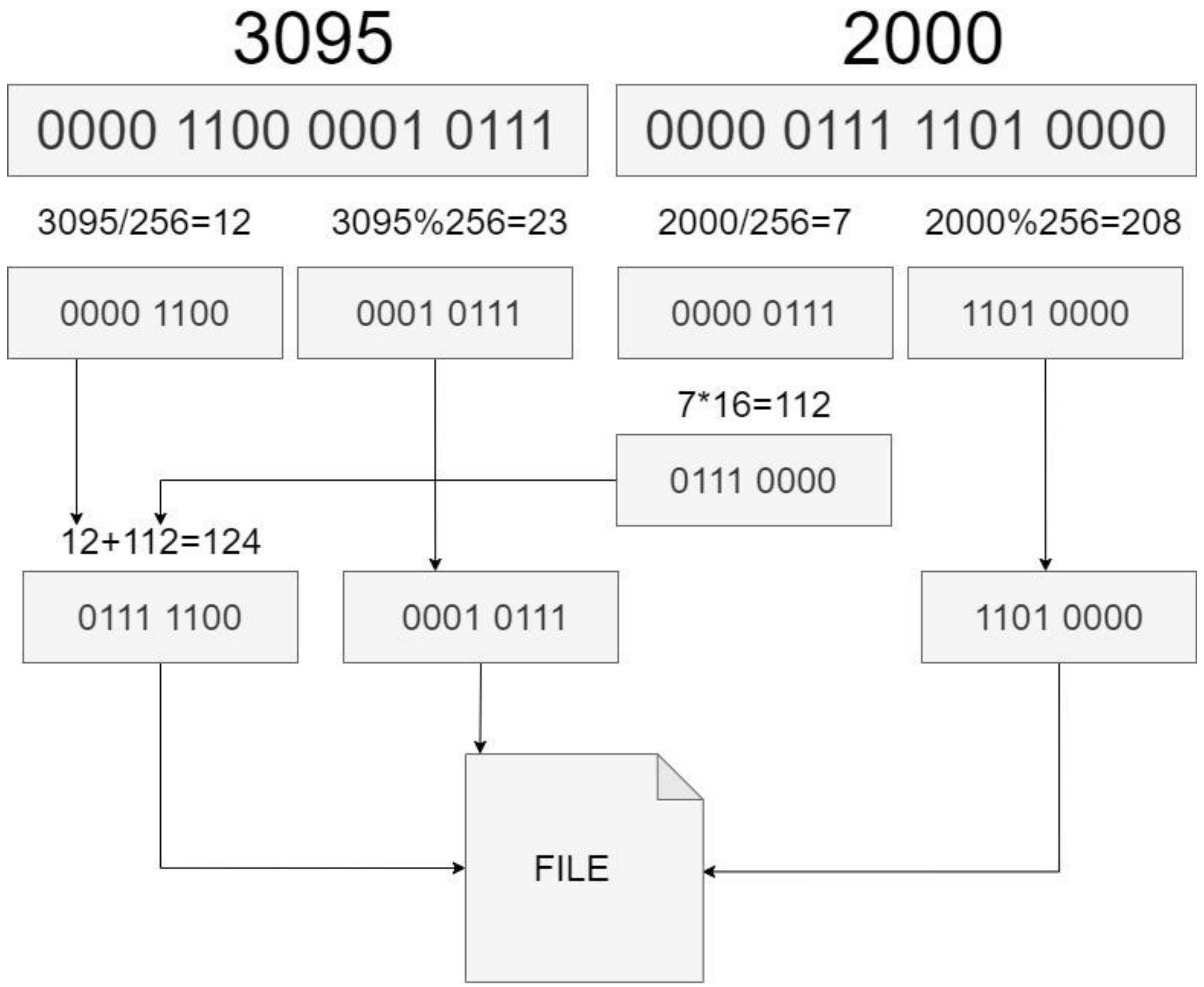
```
for i in target-current_search_buffer_length:target-1
```

For Unicode



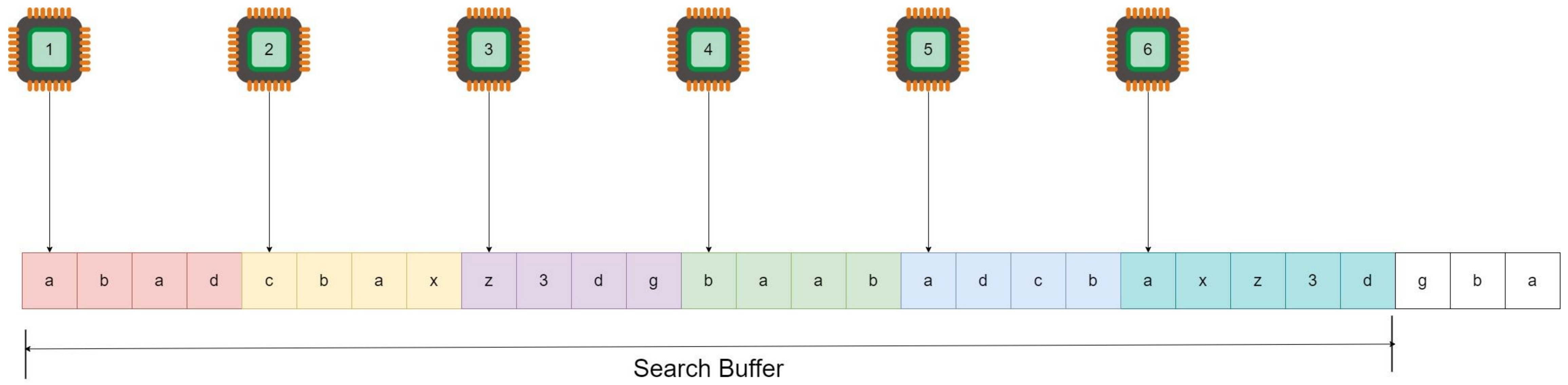
```
eachindex_correction=prevind(input,target_index,current_search_buffer_length)-1
for index in eachindex(input[prevind(input,target_index,current_search_buffer_length):prevind(input,target_index)])
```

Avoiding wastage of Space

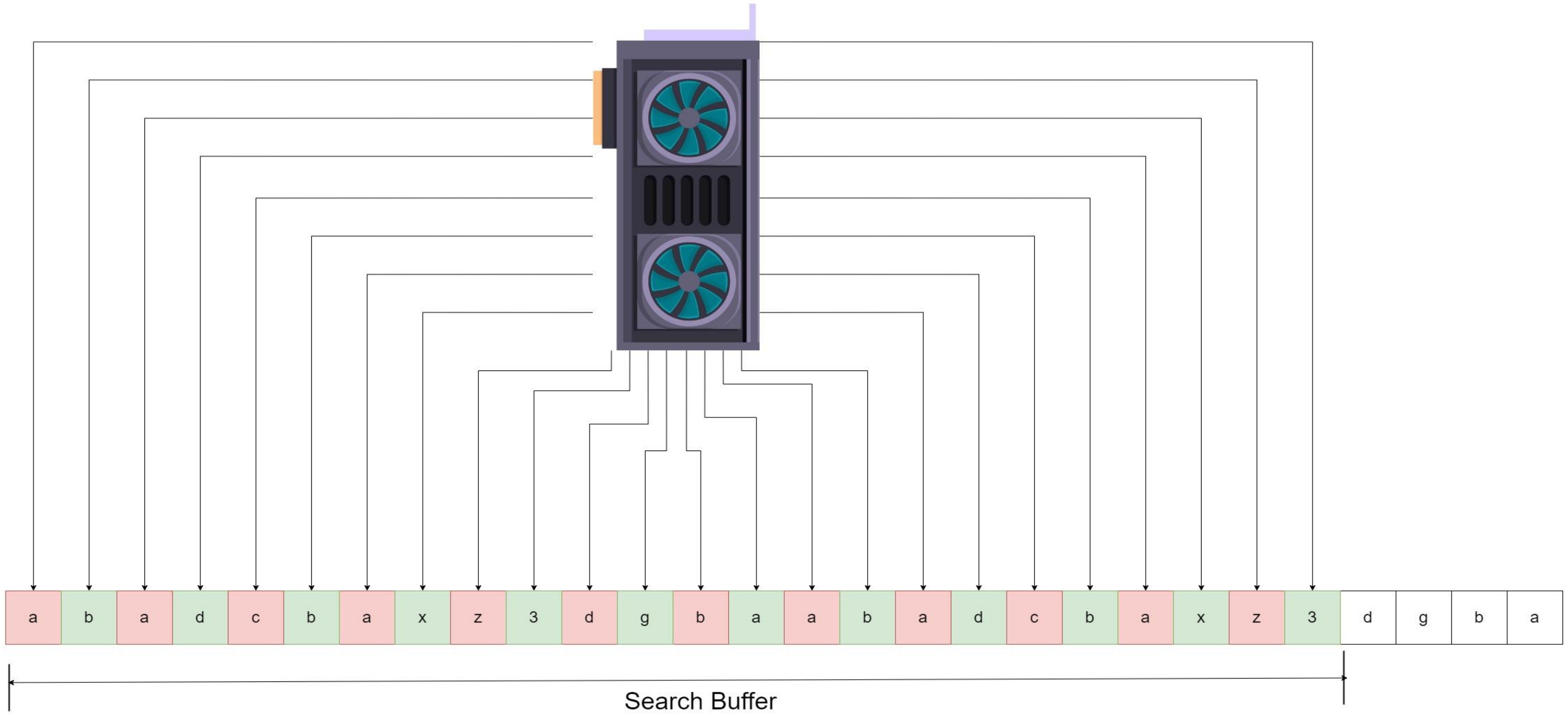


Potential Opportunities for Parallel Implementations

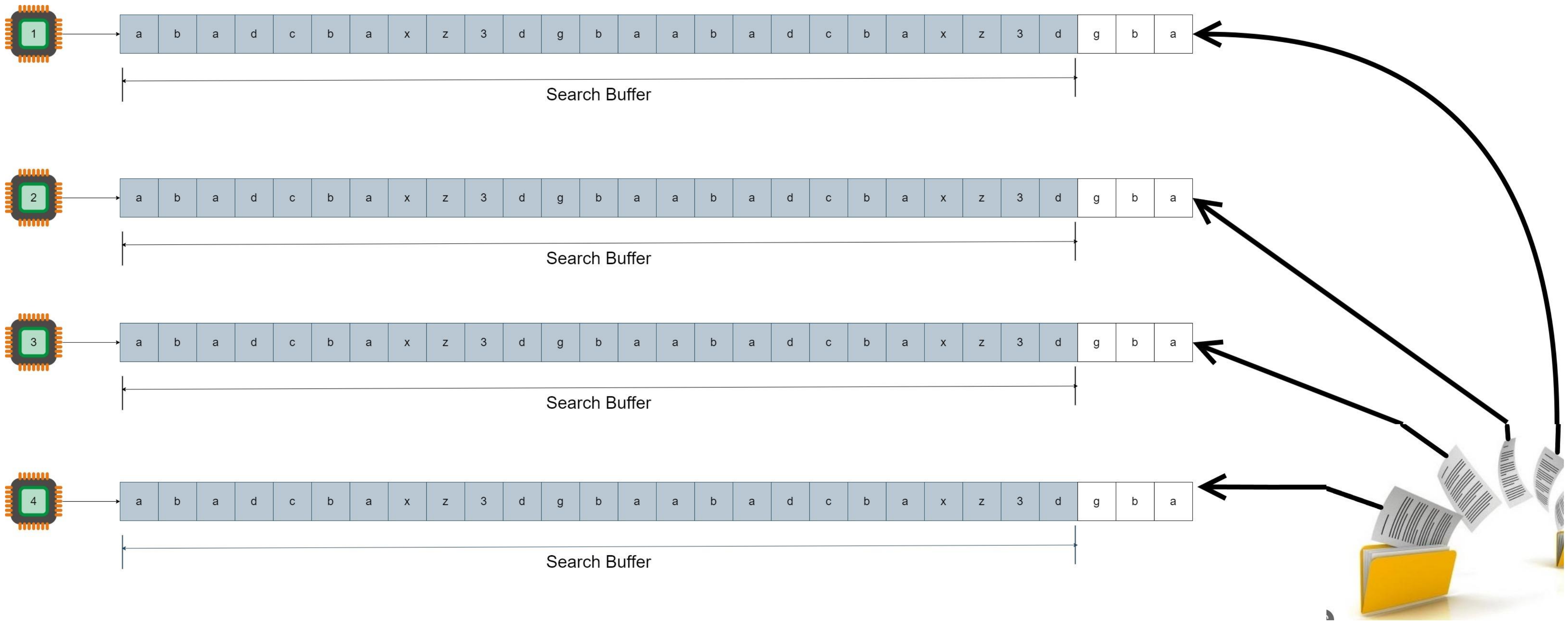
1. Parallelizing the search phase on CPU:



2. Parallelizing the search phase on GPU:



3. Dividing the File and Compressing in Parallel


1
a b a d c b a x z 3 d g b a a b a d c b a x z 3 d g b a
Search Buffer
2
a b a d c b a x z 3 d g b a a b a d c b a x z 3 d g b a
Search Buffer
3
a b a d c b a x z 3 d g b a a b a d c b a x z 3 d g b a
Search Buffer
4
a b a d c b a x z 3 d g b a a b a d c b a x z 3 d g b a
Search Buffer