TruffleStrings: a Highly Optimized Cross-Language String Implementation

Josef Haider

About Me

- Researcher at Oracle Labs and on the GraalVM Team since 2018
- Main author of:
    - TRegex, the Truffle multi-language regex engine
    - TruffleStrings, the Truffle multi-language string implementation

Motivation

- Language users expect strings to behave like in the original language implementation
- Most languages leak their internal string encoding to the user
- example: string "😕"

```
js> "\u{01F615}"
  < '\u{01F615}'

js> "\u{01F615}"[0]
  < '\uD83D'

js> "\u{01F615}".length
  < 2
```

Motivation

- Language users expect strings to behave like in the original language implementation
- Most languages leak their internal string encoding to the user

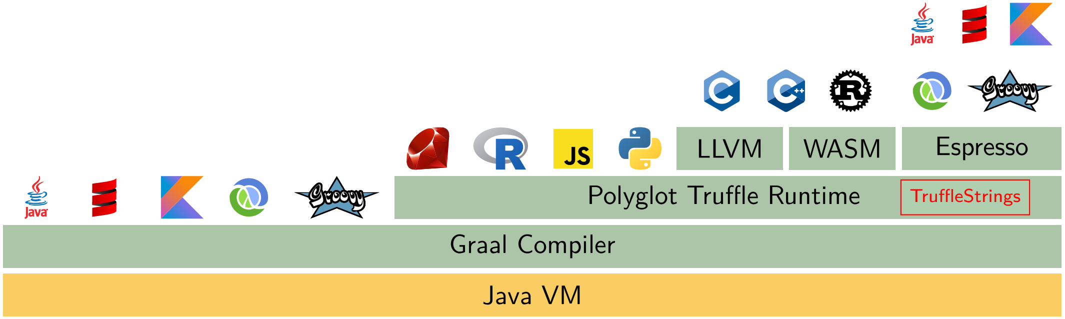
```
rb> "\u{01F615}".encoding
  < #<Encoding:UTF-8>
rb> "\u{01F615}".bytesize
  < 4
rb> "\u{01F615}".bytes
  < [240, 159, 152, 149]
rb> "\u{01F615}".each_byte.map { |b| b.to_s(16) }.join(" ")
  < "f0 9f 98 95"
```

Motivation

- Truffle treats strings as a primitive data type
- Strings may cross language boundary
    - Conversion overhead

```
let jsString = "asdf";
let rbString = callRubyFunction(jsString);
let pyString = callPythonFunction(jsString, rbString);
// ...
```

New Component: TruffleString



Requirements - Encodings

Espresso:	UTF-16
JavaScript:	UTF-16
Node.js:	UTF-8
Python:	UTF-32
R:	any (system encoding)
Ruby:	any (default: UTF-8)

Requirements - Optimizations

- Lazy concatenation
- Lazy repetition
- Lazy string from int
- Cheap conversion to and from Java String
- String views (substring without copy)
- String views into native memory (C extensions)
- String compaction
  - UTF-16:
    - LATIN-1 if all code points $\leq$ `0xff`
  - UTF-32:
    - LATIN-1 if all code points $\leq$ `0xff`
    - UCS-2 if all code points $\leq$ `0xffff`

Code Range

- Track upper limit of codepoints in strings
  - $\leq$ `0x007f`: ASCII
  - $\leq$ `0x00ff`: LATIN-1
  - $\leq$ `0xffff`: BMP
- More optimization potential on Truffle side
- Allows no-op encoding conversions
  - ASCII-only strings are equivalent in almost all encodings
  - LATIN-1 and BMP strings are equivalent in UTF-16 and UCS-2

Requirements - Ruby

- *Mutable* strings
    - Individual bytes may be overwritten
- Must track if string is ASCII-only

Polymorphism - Variable string properties

- Encoding (~100 encodings supported)
- String compaction level (3 possible states)
- managed vs native storage (Java byte array or native memory)
- immutable vs mutable (modeled as Java classes)
- lazy vs materialized (lazy concatenation, lazy int to string)

Data structure

```java
public abstract class AbstractTruffleString {
    private Object data; // byte[], NativePointer or LazyData
    private final int offset;
    private final int length;
    private final byte encoding;
    private final byte stride; // compaction level
    private final byte flags;
    int hashCode; // cache
}

public final class TruffleString extends AbstractTruffleString {
    private final int codePointLength;
    private final byte codeRange;
    private volatile TruffleString next; // transcoding cache
}

public final class MutableTruffleString extends AbstractTruffleString {
    private int codePointLength;
    private byte codeRange;
}
```

Operations

- Creating a new TruffleString
    - FromCodePoint
    - FromLong
    - FromByteArray
    - FromCharArrayUTF16
    - FromIntArrayUTF32
    - FromJavaString
    - FromNativePointer
    - Encoding.getEmpty
    - Concat
    - Substring
    - SubstringByteIndex
    - Repeat
- Comparison
    - Equal
    - RegionEqual
    - RegionEqualByteIndex
    - CompareBytes
    - CompareCharsUTF16
    - CompareIntsUTF32
    - HashCode
- Accessing codepoints and bytes
    - ReadByte
    - ReadCharUTF16
    - CodePointAtIndex
    - CodePointAtByteIndex
    - CreateCodePointIterator
    - CreateBackwardCodePointIterator
    - ByteLengthOfCodePoint
    - CodePointIndexToByteIndex
- Query string properties
    - isEmpty
    - CodePointLength
    - byteLength
    - IsValid
    - GetCodeRange
    - GetByteCodeRange
    - CodeRangeEquals
    - isCompatibleTo
    - isManaged
    - isNative
    - isImmutable
    - isMutable
- Conversion
    - SwitchEncoding
    - ForceEncoding
    - AsTruffleString
    - AsManaged
    - Materialize
    - CopyToByteArray
    - GetInternalByteArray
    - CopyToNativeMemory
    - GetInternalNativePointer
    - ToJavaString
    - ParseInt
    - ParseLong
    - ParseDouble
- Search
    - ByteIndexOfAnyByte
    - CharIndexOfAnyCharUTF16
    - IntIndexOfAnyIntUTF32
    - IndexOfCodePoint
    - ByteIndexOfCodePoint
    - LastIndexOfCodePoint
    - LastByteIndexOfCodePoint
    - IndexOfString
    - ByteIndexOfString
    - LastIndexOfString
    - LastByteIndexOfString

Optimization

- SIMD is everything
- Most string operations are very simple
- Floating-point operations: 8 single-precision or 4 double-precision values per YMM vector
- 8-bit string: **32** values per YMM vector!

Optimization

```
for (int i = 0; i < length; i++) {
    if (arrA[offA + i] != arrB[offB + i]) {
        return false;
    }
}
return true;
```

array-region-equals loop

```
movdqu ymm0, (arrayA, index)
pxor   ymm0, ymm0, (arrayB, index)
ptest  ymm0, ymm0
jnz    FalseLabel
```

SIMD loop body

Optimization

- Replace all important loops with **stub calls**
- Cheap function calls where Graal knows all clobbered registers
- Function body is **handwritten assembly**
- No safepoints, we don't have to care about the Java memory model during stub execution
- Same mechanism is used on JVM for Java String intrinsics and e.g. `System.arraycopy`

Arbitrary stride and managed/native memory

- stubs are agnostic to managed/native memory *and* compaction level

```
static int intrinsic(byte[] array, long offset, int stride)
```

```
addq array, offset
movq reg, ($jumpTable, stride)
jmp  reg
```

- native pointers are passed as `offset` with `array = null`

Intrinsified operations: copy/inflate/deflate

- Already present for Java Strings, generalized for UTF-32
- inflate: 8-bit $\rightarrow$ 16/32-bit, 16-bit $\rightarrow$ 32-bit
    - `pmovzxbw` etc.
- deflate: 32-bit $\rightarrow$ 16/8-bit, 16-bit $\rightarrow$ 8-bit
    - `packuswb` etc.

Intrinsified operations: equals/regionEquals

- pxor + ptest
- specialized versions for cases with constant stride and length

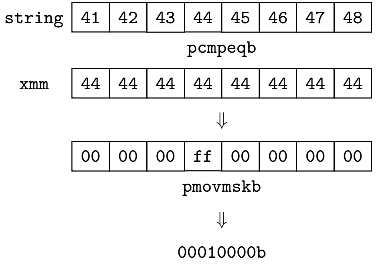
```
for (int i = 0; i < 15; i++) {
    if (arrayA[i] != arrayB[i]) {
        return false;
    }
}
return true;
```

```
movq  rax, (arrayA)
xorq  rax, (arrayB)
movq  rbx, (arrayA, 7)
xorq  rbx, (arrayB, 7)
orq   rax, rbx
jnz   FalseLabel
```

Intrinsified operations: indexOf(int)

- Previously: `pcmpestri`
- Simple AVX instructions scale better
- `pcmpeq + ptest + pmovmsk + bsfq`

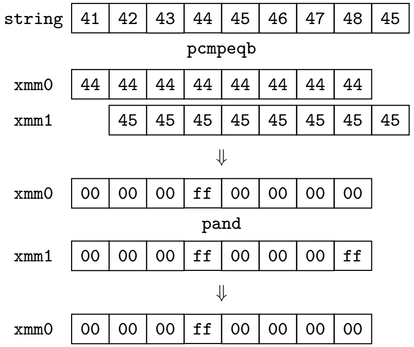
```
for (int i = 0; i < length; i++) {
    if (array[i] == value) {
        return i;
    }
}
```



Intrinsified operations: indexOf(string)

- Intrinsified version of `indexOf` for two consecutive characters
- Used in combination with `regionEquals` in a search loop

```
for (int i = 1; i < length; i++) {
    if (array[i-1] == v0 && array[i] == v1) {
        return i-1;
    }
}
```

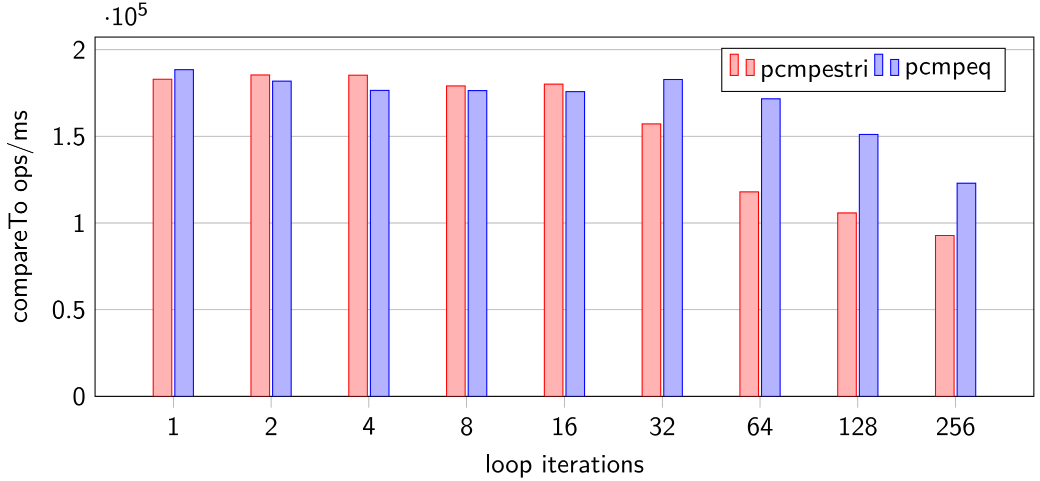

string 41 42 43 44 45 46 47 48 45
pcmpeqb
xmm0 44 44 44 44 44 44 44 44
xmm1 45 45 45 45 45 45 45 45
⇓
xmm0 00 00 00 ff 00 00 00 00
pand
xmm1 00 00 00 ff 00 00 00 ff
⇓
xmm0 00 00 00 ff 00 00 00 00

Intrinsified operations: compareTo

- Previously: `pcmpestri`
- Find index of different elements with `pcmpeq + pmovmsk + bsfq`
- Return scalar result

```
for (int i = 0; i < length; i++) {
    if (arrayA[i] != arrayB[i]) {
        return arrayA[i] - arrayB[i];
    }
}
```

Intrinsified operations: compareTo

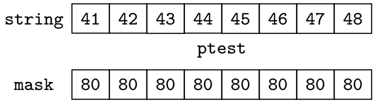


Intrinsified operations: calculate string attributes

- `calcStringAttributes` simultaneously
  - validates the string
  - calculates the number of codepoints
  - calculates the code range (rough upper bound of codepoint values)
- intrinsified for
  - US-ASCII
  - ISO-8859-1 (LATIN-1)
  - UTF-8
  - UTF-16
  - UTF-32

Intrinsified operations: calculate string attributes

- Fast path: string is ASCII-only
- Can be checked with a single `ptest` instruction!



- UTF-32: gradually loosen the `ptest` mask
  - `0xffffff80` - `0xffffff00` - `0xffff0000`

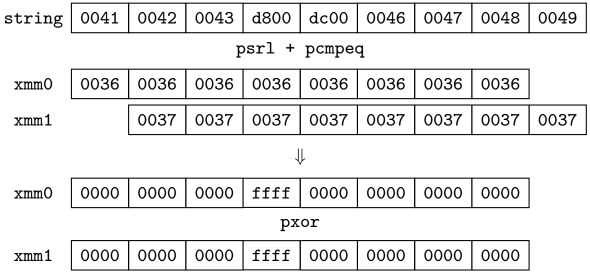
Intrinsified operations: calculate string attributes

- Validating UTF-16 surrogate pairs:

```
for (; i < length; i++) {
    if ((array[i] >> 11) == 0x1b) {
        if ((array[i] >> 10) == 0x36 && (array[i+1] >> 10) == 0x37) {
            i++;
            nCodePoints--;
        } else {
            codeRange = BROKEN;
        }
    }
}
```

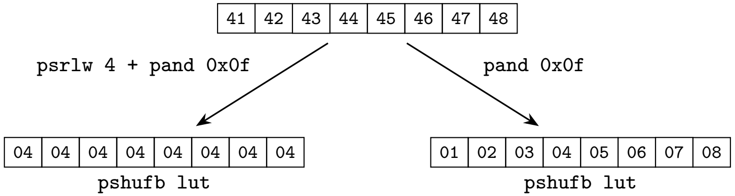
Intrinsified operations: calculate string attributes

- Validating UTF-16 surrogate pairs:
- Identify leading and trailing surrogates with `pcmpeq`
- `pxor` the result


string 0041 0042 0043 d800 dc00 0046 0047 0048 0049
psrl + pcmpeq
xmm0 0036 0036 0036 0036 0036 0036 0036 0036
xmm1 0037 0037 0037 0037 0037 0037 0037 0037
⇓
xmm0 0000 0000 0000 ffff 0000 0000 0000 0000
pxor
xmm1 0000 0000 0000 ffff 0000 0000 0000 0000

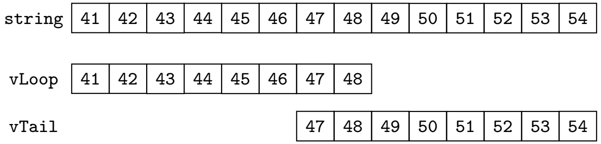
Intrinsified operations: calculate string attributes

- Validating UTF-8 strings:
- Ported algorithm from ”Validating UTF-8 In Less Than One Instruction Per Byte” by John Keiser and Daniel Lemire
- Based on lookup tables and `pshufb`



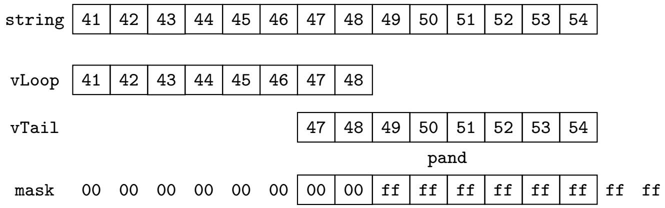
Fast tail processing

- Duplicates OK:
- Just load from `array + length - vectorSize`



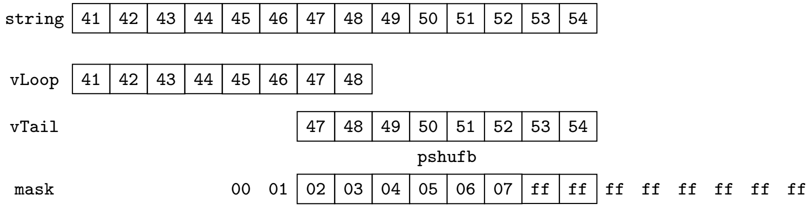
Fast tail processing

- Duplicates not OK, but zero elements don't matter:
- Remove duplicate elements with a constant mask from memory

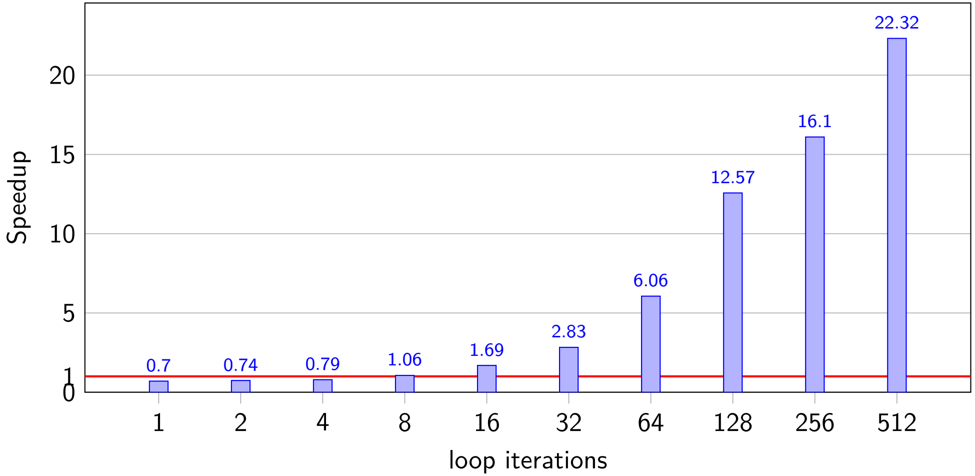

string 41 42 43 44 45 46 47 48 49 50 51 52 53 54
vLoop 41 42 43 44 45 46 47 48
vTail 47 48 49 50 51 52 53 54
pand
mask 00 00 00 00 00 00 00 00 ff ff ff ff ff ff ff ff

Fast tail processing

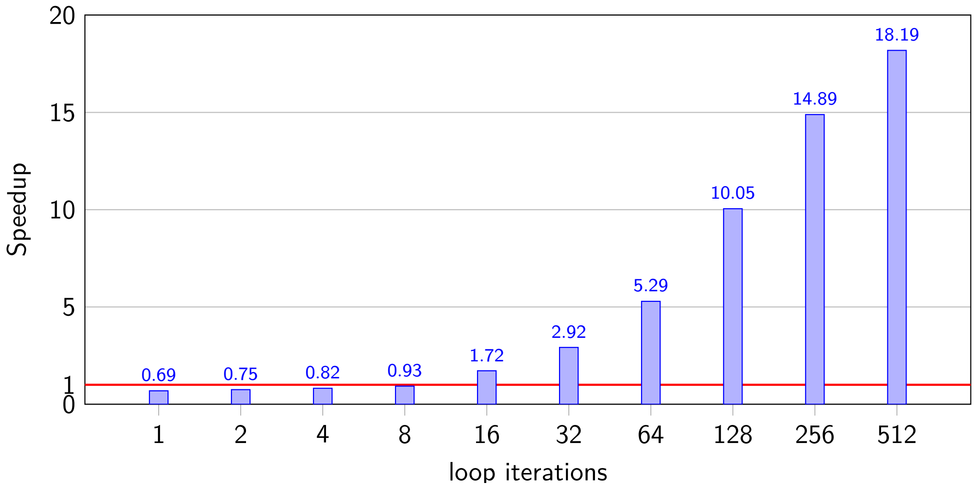
- Duplicates not OK, zero elements don't matter and order matters:
- Remove duplicate elements and reorder remaining elements wit a constant mask from memory



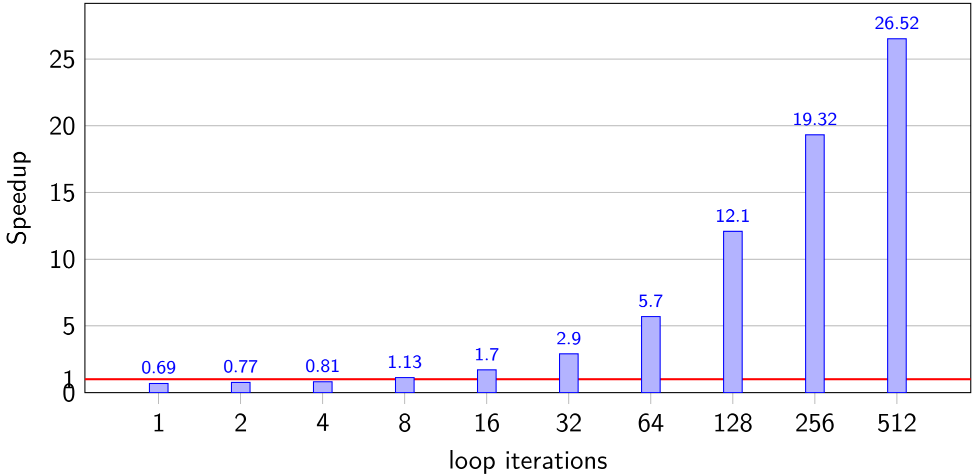
Intrinsic speedups: regionEquals



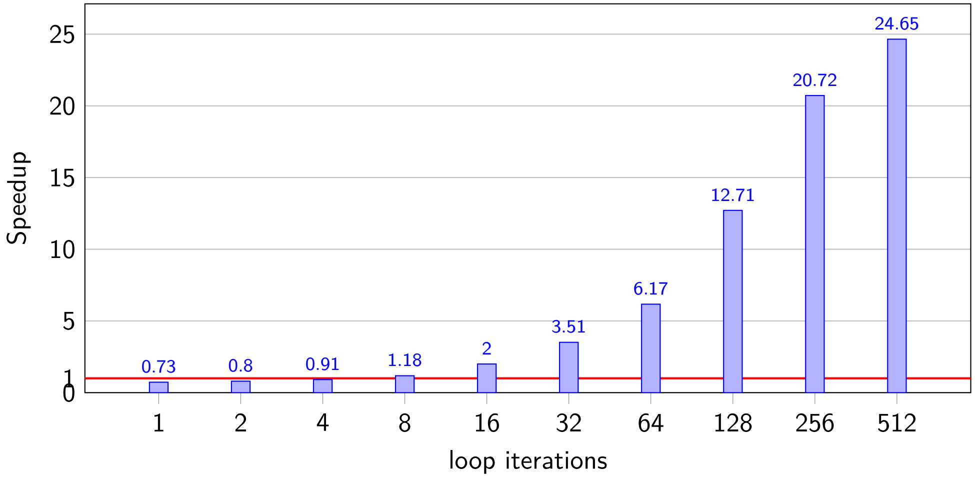
Intrinsic speedups: compareTo



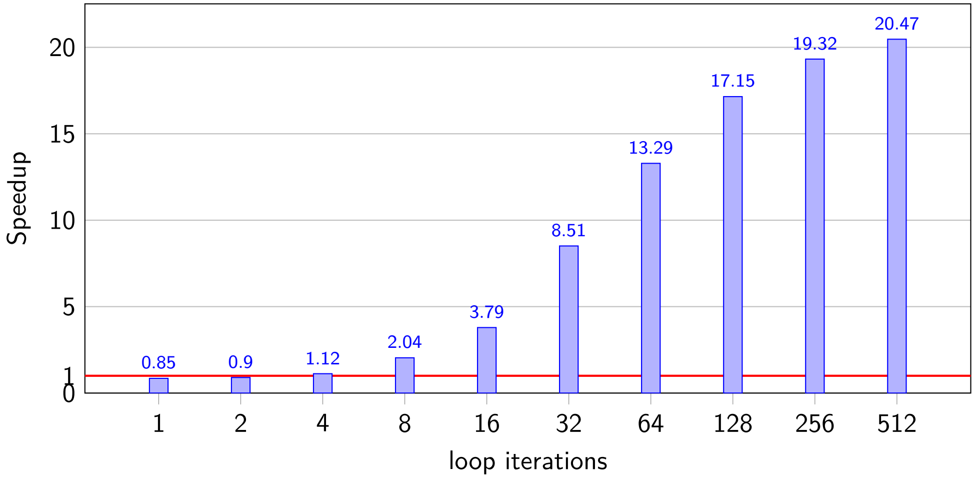
Intrinsic speedups: indexOf



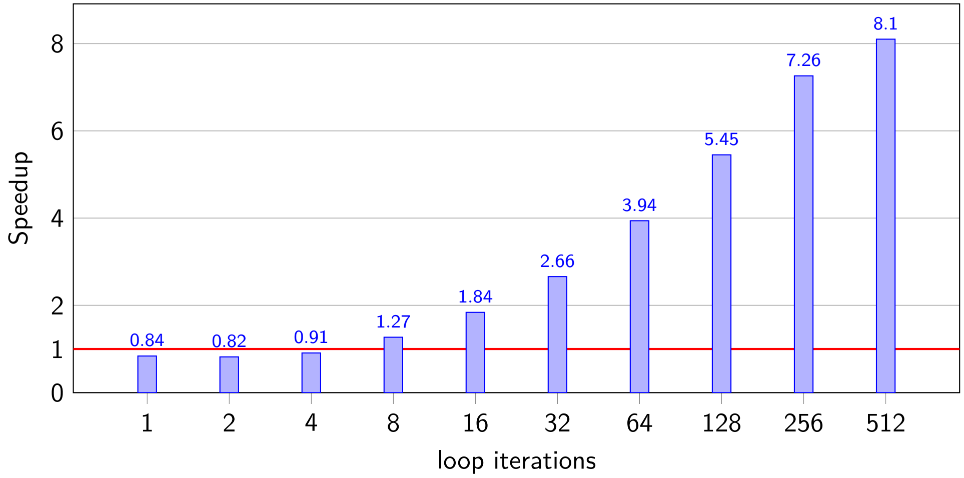
Intrinsic speedups: calcStringAttributes - ASCII



Intrinsic speedups: calcStringAttributes - UTF-8



Intrinsic speedups: calcStringAttributes - UTF-16



AARCH64 support

- Support via NEON and SVE
- Work in progress, not yet enabled
- Ported versions of all intrinsics except `calcStringAttributes` exist already for Java strings, but are missing customizations/generalizations for TruffleString

Conclusion

- TruffleString is merged already, check it out!
  - `https://github.com/oracle/graal/commit/845231e651d611ecbe5cffc0535fda0d0e83bad1`
- graal-js migrated already
- truffleruby and graalpython migration is in progress