# What do Intel's New AVX-512 Instructions Mean for High-Performance Unicode?

Robert D. Cameron

School of Computing Science
Simon Fraser University

September 11, 2018

# Outline


1. Introduction to SIMD/AVX-512
2. Parabix: Scalable High-Performance Unicode
3. Bitwise Data Parallel Regular Expression Matching
4. Programming Framework: Kernels + Stream Sets = Programs
5. `icgrep` Architecture
6. Scalable Performance Results
7. Conclusion

# Outline


1. Introduction to SIMD/AVX-512
2. Parabix: Scalable High-Performance Unicode
3. Bitwise Data Parallel Regular Expression Matching
4. Programming Framework: Kernels + Stream Sets = Programs
5. icgrep Architecture
6. Scalable Performance Results
7. Conclusion

# What Are Vector Instructions?

## Single Instruction Multiple Data (SIMD)

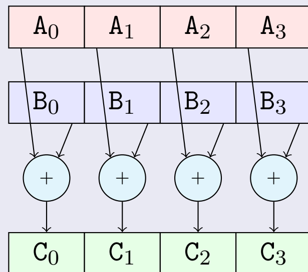
- Vector instructions implement the SIMD concept.
- Multiple vector elements are combined in a single operation.

# What Are Vector Instructions?

## Single Instruction Multiple Data (SIMD)

- Vector instructions implement the SIMD concept.
- Multiple vector elements are combined in a single operation.

## Example: <4 x i16> Vector Addition




- Four 16-bit additions at once.
- Replaces the loop:

```
for (int i=0; i<4; i++) {
  C[i] = A[i] + B[i];
}
```

- Intel `paddw` MMX instruction.

# Two Decades of Intel SIMD Extensions

## MMX, SSE, AVX, AVX-512

- 1997: Intel Pentium processors introduce 64-bit SIMD (MMX).
- 1999: 128-bit floating point SIMD vectors (SSE).
- 2000: SSE2 adds 128-bit integer vector operations.
  - Widespread Intel/AMD standard: all x86-64 processors.
- SSE3 (2004), SSSE3 (2005), SSE4 (2008).
- 256-bit SIMD: Intel AVX (2011), AVX2 (2013).
- 512-bit SIMD: Intel AVX-512 (2017)

# Two Decades of Intel SIMD Extensions

## MMX, SSE, AVX, AVX-512

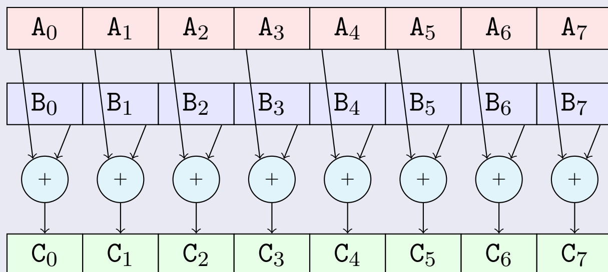
- 1997: Intel Pentium processors introduce 64-bit SIMD (MMX).
- 1999: 128-bit floating point SIMD vectors (SSE).
- 2000: SSE2 adds 128-bit integer vector operations.
  - Widespread Intel/AMD standard: all x86-64 processors.
- SSE3 (2004), SSSE3 (2005), SSE4 (2008).
- 256-bit SIMD: Intel AVX (2011), AVX2 (2013).
- 512-bit SIMD: Intel AVX-512 (2017)

## Not Just Intel/AMD

- Altivec/VMX (2004): 128-bit SIMD
- ARM Neon (2008): 128-bit SIMD
- ARM SVE: scalable SIMD up to 2048 bits (in progress)

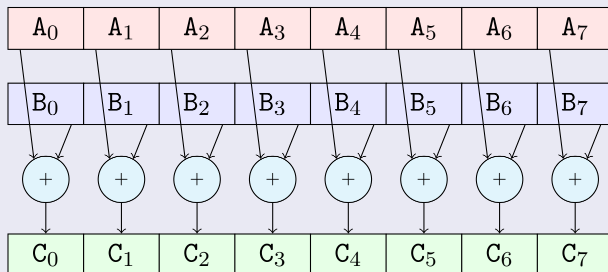
# Doubling Register Size, Repeatedly

## 128-bit SSE2: <8 x i16> Vector Addition



# Doubling Register Size, Repeatedly

## 128-bit SSE2: <8 x i16> Vector Addition




## AVX2, AVX-512

- 256-bit AVX2: `<16 x i16>` vector addition in a single operation.
- 512-bit AVX-512: `<32 x i16>` vector addition in a single operation.

# Why SIMD?

## SIMD Costs

- Intel has added hundreds of SIMD instructions over two decades.
  - More SIMD instructions added than any other kind.
  - Substantial transistor count, chip area devoted to SIMD.
  - Larger cores, so fewer cores per package possible.

# Why SIMD?

## SIMD Costs

- Intel has added hundreds of SIMD instructions over two decades.
  - More SIMD instructions added than any other kind.
  - Substantial transistor count, chip area devoted to SIMD.
  - Larger cores, so fewer cores per package possible.

## SIMD Benefits

- SIMD naturally supports data parallel applications.
  - Graphics, signal and image processing.
  - Physical simulation.
  - Database queries, financial analytics.
- SIMD has natural advantages over multicore.
  - Cost of instruction fetch/decode divided by SIMD vector length.
  - SIMD ALUs share common control and data path logic.
  - Synchronization of parallel execution is automatic.

# What's new in AVX-512?

## Several New Instruction Families

- AVX-512F: Foundation - core 32/64 bit operations (extending AVX).
- AVX-512DQ: New doubleword/quadword (32/64-bit) operations.
- AVX-512BW: AVX-2 byte/word operations extended to 512 bits.
- AVX-512VBMI: Full byte-level permutation selecting from 128 bytes.
- Several additional small families of specialized instructions.

# What's new in AVX-512?

## Several New Instruction Families

- AVX-512F: Foundation - core 32/64 bit operations (extending AVX).
- AVX-512DQ: New doubleword/quadword (32/64-bit) operations.
- AVX-512BW: AVX-2 byte/word operations extended to 512 bits.
- AVX-512VBMI: Full byte-level permutation selecting from 128 bytes.
- Several additional small families of specialized instructions.

## Systematic New Features

- Systematic masking and blending using bitmask registers.
- Constant parameter broadcasting, rounding and exception control.
- Register count increased from 16 to 32.
- Ternary logic - all possible 3-bit Boolean functions.

# Opportunity and Challenge

## Opportunity

- Extensive SIMD parallelism offers the potential to dramatically speed-up applications.
- The expected speed-up is potentally very large.
- Considerable data rearrangement overhead can be tolerated.

# Opportunity and Challenge

## Opportunity

- Extensive SIMD parallelism offers the potential to dramatically speed-up applications.
- The expected speed-up is potentally very large.
- Considerable data rearrangement overhead can be tolerated.

## Challenge

- Existing sequential programs generally cannot be autovectorized.
  - Too many sequential dependencies between data elements.
  - Programmer code optimizations often obscure parallelizable logic.
- Language technology may limit access to SIMD capabilities.
- Text processing may involve variable-length items not well-matched to fixed SIMD field and register widths.
- Data parallel algorithmic approaches may be hard to find.

# Outline


1. Introduction to SIMD/AVX-512
2. Parabix: Scalable High-Performance Unicode
3. Bitwise Data Parallel Regular Expression Matching
4. Programming Framework: Kernels + Stream Sets = Programs
5. `icgrep` Architecture
6. Scalable Performance Results
7. Conclusion

# Parabix Technology

## Parabix Concept

- Programming framework for high-performance data stream processing.
- Employs novel algorithms based on *bitwise data parallelism*.
    - Process 128 bytes at a time using 128 bit registers (SSE2).
- Fully utilizes processor wide vector instructions (SIMD).

# Parabix Technology

## Parabix Concept

- Programming framework for high-performance data stream processing.
- Employs novel algorithms based on *bitwise data parallelism*.
    - Process 128 bytes at a time using 128 bit registers (SSE2).
- Fully utilizes processor wide vector instructions (SIMD).

## Parabix Scalability

- Parabix scales to use available SIMD register width.
    - Intel AVX2 (2013): 256 bytes at a time.
    - Intel AVX-512 (2017): 512 bytes at a time.
- Parabix can also scale to use multiple cores, even on a single data stream.
- No changes to application programs required!

# Regular Expression Showcase: icgrep

## icgrep 1.8

- Full-featured grep implementation using Parabix algorithms.
- Posix REs: Basic or Extended
  - All features except backreferences.
- Perl-compatible REs (PCRE)

# Regular Expression Showcase: icgrep

## icgrep 1.8

- Full-featured grep implementation using Parabix algorithms.
- Posix REs: Basic or Extended
  - All features except backreferences.
- Perl-compatible REs (PCRE)

## UTS #18 - Unicode Regular Expressions

- Full Unicode property support.
- Set operations, e.g., `[\p{Greek}&&\p{upper case}]`
- Grapheme clusters and grapheme cluster mode.
- Name property with regexp values `\p{name=/AIRPLANE/}`
- Canonical and compatible equivalence.

# Outline

1. Introduction to SIMD/AVX-512
2. Parabix: Scalable High-Performance Unicode
3. Bitwise Data Parallel Regular Expression Matching
4. Programming Framework: Kernels + Stream Sets = Programs
5. icgrep Architecture
6. Scalable Performance Results
7. Conclusion

# Beyond Byte-At-A-Time

- Traditional regular expression technology processes one code unit at a time using DFA, NFA or backtracking implementations.
- Instead consider a bitwise data parallel approach.
- Byte-oriented data is first transformed to 8 parallel bit streams (Parabix transform).
- Bit stream $j$ consists of bit $j$ of each byte.
- Load 128-bit SIMD registers to process 128 positions at a time in bitwise data parallel fashion (SSE2, ARM Neon, ...).
- Or use 256-bit AVX2 registers of newer Intel processors.
- Process using bitwise logic, shifting and addition.
- Parabix methods have previously been used to accelerate Unicode transcoding and XML parsing.

# Unbounded Stream Abstraction

- Program operations as if *all positions in the file are to be processed simultaneously*.
- Unbounded bitwise parallelism.
- Pablo compiler technology maps to block-by-block processing.
- Information flows between blocks using carry bits.
- LLVM compiler infrastructure for Just-in-Time compilation.
- Custom LLVM improvements further accelerate processing.

# Marker Streams

- Marker stream $M_i$ indicates the positions that are reachable after item $i$ in the regular expression.
- Each marker stream $M_i$ has one bit for every input byte in the input file.
- $M_i[j] = 1$ if and only if a match to the regular expression up to and including item $i$ in the expression occurs at position $j - 1$ in the input stream.
- Conceptually, marker streams are computed in parallel for all positions in the file at once (bitwise data parallelism).
- In practice, marker streams are computed block-by-block, where the block size is the size of a SIMD register in bits.

# Marker Stream Example

- Consider matching regular expression `a[0-9]*[z9]` against the input text below.

input data `a453z--b3z--az--a12949z--ca22z7--`

# Marker Stream Example

- Consider matching regular expression `a[0-9]*[z9]` against the input text below.
- $M_1$ marks positions after occurrences of a.

```
input data   a453z--b3z--az--a12949z--ca22z7--
    M1       .1...........1...1.........1.....
```

# Marker Stream Example

- Consider matching regular expression `a[0-9]*[z9]` against the input text below.
- $M_1$ marks positions after occurrences of `a`.
- $M_2$ marks positions after occurrences of `a[0-9]*`.

| | |
|---|---|
| input data | `a453z--b3z--az--a12949z--ca22z7--` |
| $M_1$ | `.1...........1...1.........1.....` |
| $M_2$ | `.1111........1...111111....111...` |

# Marker Stream Example

- Consider matching regular expression `a[0-9]*[z9]` against the input text below.
- $M_1$ marks positions after occurrences of `a`.
- $M_2$ marks positions after occurrences of `a[0-9]*`.
- $M_3$ marks positions after occurrences of `a[0-9]*[z9]`.

| | |
|---|---|
| input data | `a453z--b3z--az--a12949z--ca22z7--` |
| $M_1$ | `.1............1..1.........1.....` |
| $M_2$ | `.1111.........1..111111....111...` |
| $M_3$ | `.....1.........1....1.11......1..` |

# Matching Character Class Repetitions with MatchStar

# Matching Character Class Repetitions with MatchStar

- MatchStar$(M, C) = (((M \wedge C) + C) \oplus C) \vee M$

# Matching Character Class Repetitions with MatchStar

- MatchStar$(M, C) = (((M \wedge C) + C) \oplus C) \vee M$
- Consider $M_2 =$ MatchStar$(M_1, C)$

| | |
|---|---|
| input data | `a453z--b3z--az--a12949z--ca22z7--` |
| $M_1$ | `.1...........1...1.........1.....` |
| $C =$ [0-9] | `.111....1........11111.....11.1..` |

# Matching Character Class Repetitions with MatchStar

- MatchStar$(M, C) = (((M \wedge C) + C) \oplus C) \vee M$
- Consider $M_2 =$ MatchStar$(M_1, C)$
- Use addition to scan each marker through the class.

| | |
|---|---|
| input data | `a453z--b3z--az--a12949z--ca22z7--` |
| $M_1$ | `.1...........1...1.........1.....` |
| $C =$ [0-9] | `.111....1........11111.....11.1..` |
| $T_0 = M_1 \wedge C$ | `.1...............1.........1.....` |
| $T_1 = T_0 + C$ | `....1...1.............1......11..` |

# Matching Character Class Repetitions with MatchStar

- MatchStar$(M, C) = (((M \wedge C) + C) \oplus C) \vee M$
- Consider $M_2 =$ MatchStar$(M_1, C)$
- Use addition to scan each marker through the class.
- Bits that change represent matches.

| | |
|---|---|
| input data | `a453z--b3z--az--a12949z--ca22z7--` |
| $M_1$ | `.1...........1...1.........1.....` |
| $C =$ [0-9] | `.111....1........11111.....11.1..` |
| $T_0 = M_1 \wedge C$ | `.1...............1.........1.....` |
| $T_1 = T_0 + C$ | `....1...1.............1......11..` |
| $T_2 = T_1 \oplus C$ | `.1111............111111....111...` |

# Matching Character Class Repetitions with MatchStar

- MatchStar$(M, C) = (((M \wedge C) + C) \oplus C) \vee M$
- Consider $M_2 =$ MatchStar$(M_1, C)$
- Use addition to scan each marker through the class.
- Bits that change represent matches.
- We also have matches at start positions in $M_1$.

| | |
|---|---|
| input data | `a453z--b3z--az--a12949z--ca22z7--` |
| $M_1$ | `.1...........1...1.........1.....` |
| $C =$ [0-9] | `.111....1........11111.....11.1..` |
| $T_0 = M_1 \wedge C$ | `.1...............1.........1.....` |
| $T_1 = T_0 + C$ | `....1...1.............1......11..` |
| $T_2 = T_1 \oplus C$ | `.1111............111111....111...` |
| $M_2 = T_2 \vee M_1$ | `.1111........1...111111....111...` |

# Matching Equations

The rules for bitwise data parallel regular expression matching can be summarized by these equations.

$$
\begin{aligned}
\text{Match}(m, C) &= \text{Advance}(\text{CharClass}(C) \wedge m) \\
\text{Match}(m, RS) &= \text{Match}(\text{Match}(m, R), S) \\
\text{Match}(m, R|S) &= \text{Match}(m, R) \vee \text{Match}(m, S)) \\
\text{Match}(m, C*) &= \text{MatchStar}(m, \text{CharClass}(C)) \\
\text{Match}(m, R*) &= m \vee \text{Match}(\text{Match}(m, R), R*) \\
\text{Advance}(m) &= m + m \\
\text{MatchStar}(m, C) &= (((m \wedge C) + C) \oplus C) \vee m
\end{aligned}
$$

The recursive equation is implemented with a while loop.

# Outline

1. Introduction to SIMD/AVX-512
2. Parabix: Scalable High-Performance Unicode
3. Bitwise Data Parallel Regular Expression Matching
4. Programming Framework: Kernels + Stream Sets = Programs
5. icgrep Architecture
6. Scalable Performance Results
7. Conclusion

# Stream Sets and Buffers

## Stream Sets

- A stream set type is of the form $N \times \texttt{iK}$
- $N$ streams of items, each item of width $K = 2^k$ bits
- All streams in a set are of the same length $L$ (may be unknown).

# Stream Sets and Buffers

## Stream Sets

- A stream set type is of the form $N \times$ `iK`
- $N$ streams of items, each item of width $K = 2^k$ bits
- All streams in a set are of the same length $L$ (may be unknown).

## Buffers

- Buffers are storage for *segments* of stream sets.
- All of the streams of a set are stored in a single buffer.
- Stream sets are stored block-at-a-time (significant for $N > 1$)
- Different buffering strategies.
  - Full stream length (mmap)
  - Fixed length circular buffer.
  - Fixed length buffer with copyback.
  - Expanding buffer (expands as needed).

# Kernels

## Kernel Structure

- Kernels are computational abstractions for text stream processing.
- Kernels process input stream sets, producing output stream sets.

# Kernels

## Kernel Structure

- Kernels are computational abstractions for text stream processing.
- Kernels process input stream sets, producing output stream sets.

## Transposition Kernel

- Input: $1 \times$ `i8`: a single stream of 8-bit code units (e.g., UTF-8).
- Output: $8 \times$ `i1`: a set 8 of parallel bit streams (basis bit streams).

# Kernels

## Kernel Structure

- Kernels are computational abstractions for text stream processing.
- Kernels process input stream sets, producing output stream sets.

## Transposition Kernel

- Input: $1 \times$ `i8`: a single stream of 8-bit code units (e.g., UTF-8).
- Output: $8 \times$ `i1`: a set 8 of parallel bit streams (basis bit streams).

## Transposition Subkernels

- Transposition can actually be divided into 3 stages.
- Stage 1: $1 \times$ `i8`: to $2 \times$ `i4` (2 streams of nybbles).
- Stage 2: $2 \times$ `i4`: to $4 \times$ `i2` (4 streams of bit-pairs).
- Stage 3: $4 \times$ `i2`: to $8 \times$ `i1` (basis bit streams).

# Regular Expression Kernels

## Character Class Kernels

- Kernel for the character classes of a regexp: e.g., `a[0-9]*[z9]`
- Input: $8 \times$ `i1`: the 8 basis bit streams.
- Output: $3 \times$ `i1`: 3 bit streams for `[a]`, `[0-9]`, `[z9]`
- Dynamically generated by the Parabix character class compiler (ccc).

# Regular Expression Kernels

## Character Class Kernels

- Kernel for the character classes of a regexp: e.g., `a[0-9]*[z9]`
- Input: $8 \times$ `i1`: the 8 basis bit streams.
- Output: $3 \times$ `i1`: 3 bit streams for `[a]`, `[0-9]`, `[z9]`
- Dynamically generated by the Parabix character class compiler (ccc).

## Matching Logic Kernels

- Kernel for the matching logic: e.g., `a[0-9]*[z9]`
- Input: $3 \times$ `i1`: character class streams
- Output: $1 \times$ `i1`: a bit stream of matches found.
- Dynamically generated by the Parabix Regular Expression compiler.

# Modular icgrep Kernels

## Line Break Kernel

- Kernel for Unicode line breaks
- Input: $8 \times$ `i1`: the 8 basis bit streams.
- Output: $1 \times$ `i1`: line breaks for any of LF, CR, CRLF, LS, PS, $\cdots$

# Modular icgrep Kernels

## Line Break Kernel

- Kernel for Unicode line breaks
- Input: $8 \times$ `i1`: the 8 basis bit streams.
- Output: $1 \times$ `i1`: line breaks for any of LF, CR, CRLF, LS, PS, $\cdots$

## Match Scanning Kernel

- Kernel to generate matched lines.
- Three inputs:
  - $1 \times$ `i8`: source byte stream
  - $1 \times$ `i1`: matches bit stream
  - $1 \times$ `i1`: line break bit stream
- Output: $1 \times$ `i8` matched line output stream.

# Kernel Composition: Pipelines

## Kernels + StreamSets = Programs

- Name the stream sets used as inputs and outputs to each kernel.
- Compose a program as a sequence of kernels.

# Kernel Composition: Pipelines

## Kernels + StreamSets = Programs

- Name the stream sets used as inputs and outputs to each kernel.
- Compose a program as a sequence of kernels.

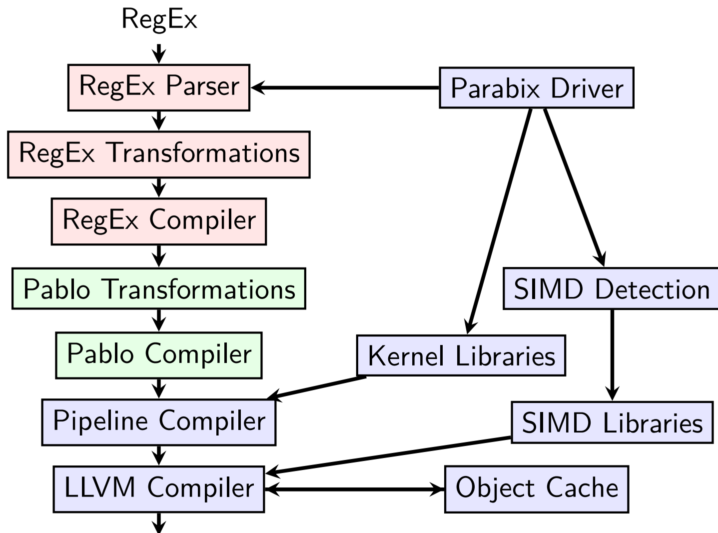
## A 7-Stage icgrep Program

```
          ByteData = MMapSource(FileName)
         BasisBits = Transpose(ByteData)
          LineEnds = UnicodeLineBreaks(BasisBits)
  CharacterClasses = CC_compiler<regexp>(BasisBits)
           Matches = RE_compiler<regexp>(CharacterClasses)
      MatchedLines = MatchScanner(ByteData, LineEnds, Matches)
                     StdoutSink(MatchedLines)
```

# Outline

1. Introduction to SIMD/AVX-512
2. Parabix: Scalable High-Performance Unicode
3. Bitwise Data Parallel Regular Expression Matching
4. Programming Framework: Kernels + Stream Sets = Programs
5. `icgrep` Architecture
6. Scalable Performance Results
7. Conclusion

# Parabix Compilation Architecture: icgrep



# Outline


1. Introduction to SIMD/AVX-512
2. Parabix: Scalable High-Performance Unicode
3. Bitwise Data Parallel Regular Expression Matching
4. Programming Framework: Kernels + Stream Sets = Programs
5. icgrep Architecture
6. Scalable Performance Results
7. Conclusion

# Scalability in Simple String Search

## Example: Search for the string "grep"

- Data source: 620 MB Wikibooks document set (15 languages)
- Boyer-Moore allows grep to skip characters, but IPC poor.
- icgrep/SSE2 not much faster, but scales up with AVX.

# Scalability in Simple String Search

## Example: Search for the string "grep"

- Data source: 620 MB Wikibooks document set (15 languages)
- Boyer-Moore allows grep to skip characters, but IPC poor.
- icgrep/SSE2 not much faster, but scales up with AVX.

## Performance Results

| Program | Processor | SIMD | Instructions | Time |
|---|---|---|---|---|
| grep | i7-3770 @ 3.4 GHz | SSE2 | 758 M | 0.37 s |
| | i3-5010U @ 2.1 GHz | AVX2 | 757 M | 0.54 s |
| | W-2102 @ 2.9 GHz | AVX-512 | 756 M | 0.44 s |
| icgrep | i7-3770 @ 3.4 GHz | SSE2 | 1,515 M | 0.30 s |
| | i3-5010U @ 2.1 GHz | AVX2 | 903 M | 0.26 s |
| | W-2102 @ 2.9 GHz | AVX-512 | 641 M | 0.18 s |
| | W-2102 (2 cores) | AVX-512 | 648 M | 0.12 s |

# Case-Insensitive String Search: grep vs. icgrep

## Example: Search for the string "find"

Command flag: `-i` Regex: `find`

- Data source: 620 MB Wikibooks document set (15 languages)

# Case-Insensitive String Search: grep vs. icgrep

## Example: Search for the string "find"

Command flag: `-i` Regex: `find`

- Data source: 620 MB Wikibooks document set (15 languages)

## Performance Results

| Program | Processor | SIMD | Instructions | Time |
|---|---|---|---|---|
| grep -i | i7-3770 @ 3.4 GHz | SSE2 | 4,454 M | 1.07 s |
| | i3-5010U @ 2.1 GHz | AVX2 | 4,454 M | 1.66 s |
| | W-2102 @ 2.9 GHz | AVX-512 | 4,453M | 1.41 s |
| icgrep -i | i7-3770 @ 3.4 GHz | SSE2 | 3,221 M | 0.42 s |
| | i3-5010U @ 2.1 GHz | AVX2 | 1,860 M | 0.43 s |
| | W-2102 @ 2.9 GHz | AVX-512 | 1,181 M | 0.28 s |
| | W-2102 (2 cores) | AVX-512 | 1,191 M | 0.16 s |

# Unicode Categories: grep vs. icgrep

## Example: Upper Case Cyrillic

Regex: `[\p{Cyrillic}&&\p{Lu}]`

grep (PCRE mode) alternative: `\p{Cyrillic}(?<=\p{Lu})`

- Data source: 620 MB Wikibooks document set (15 languages)

# Unicode Categories: grep vs. icgrep

## Example: Upper Case Cyrillic

Regex: `[\p{Cyrillic}&&\p{Lu}]`

grep (PCRE mode) alternative: `\p{Cyrillic}(?<=\p{Lu})`

- Data source: 620 MB Wikibooks document set (15 languages)

## Performance Results

| Program | Processor | SIMD | Instructions | Time |
|---|---|---|---|---|
| grep -P | i7-3770 @ 3.4 GHz | SSE2 | 2,191,635 M | 232.3 s |
| | i3-5010U @ 2.1 GHz | AVX2 | 2,191,744 M | 348.0 s |
| | W-2102 @ 2.9 GHz | AVX-512 | 2,191,552 M | 220.8 s |
| icgrep | i7-3770 @ 3.4 GHz | SSE2 | 6,678 M | 0.85 s |
| | i3-5010U @ 2.1 GHz | AVX2 | 3,683 M | 0.84 s |
| | W-2102 @ 2.9 GHz | AVX-512 | 2,174 M | 0.44 s |
| | W-2102 (2 cores) | AVX-512 | 2,206 M | 0.25 s |

# Large Bounded Repetitions

## Example: Lines >= 400 Characters

Regex: `.{400}`

- Data source: 620 MB Wikibooks document set (15 languages)
- icgrep has $\log_2$ algorithm.

# Large Bounded Repetitions

## Example: Lines >= 400 Characters

Regex: `.{400}`

- Data source: 620 MB Wikibooks document set (15 languages)
- icgrep has $\log_2$ algorithm.

## Performance Results

| Program | Processor | SIMD | Instructions | Time |
|---|---|---|---|---|
| grep -E | i7-3770 @ 3.4 GHz | SSE2 | 2,372,838 M | 249.9 s |
| | i3-5010U @ 2.1 GHz | AVX2 | 2,354,380 M | 407.8 s |
| | W-2102 @ 2.9 GHz | AVX-512 | 2,354,065 M | 247.1 s |
| icgrep | i7-3770 @ 3.4 GHz | SSE2 | 17,410 M | 2.34 s |
| | i3-5010U @ 2.1 GHz | AVX2 | 7,938 M | 1.93 s |
| | W-2102 @ 2.9 GHz | AVX-512 | 15,135 M | 2.41 s |
| | W-2102 (2 cores) | AVX-512 | 15,268 M | 1.27 s |

# Nondeterministic Matching

## Example: IP address regex

```
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
        (\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)){3}
```

- Data source: 620 MB Wikibooks document set (15 languages)

# Nondeterministic Matching

## Example: IP address regex

```
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
        (\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)){3}
```

- Data source: 620 MB Wikibooks document set (15 languages)

## Performance Results

| Program | Processor | SIMD | Instructions | Time |
|---|---|---|---|---|
| grep -E | i7-3770 @ 3.4 GHz | SSE2 | 232,079 M | 21.3 s |
| | i3-5010U @ 2.1 GHz | AVX2 | 232,423 M | 39.5 s |
| | W-2102 @ 2.9 GHz | AVX-512 | 232,081 M | 25.6 s |
| icgrep | i7-3770 @ 3.4 GHz | SSE2 | 3,720 M | 0.49 s |
| | i3-5010U @ 2.1 GHz | AVX2 | 2,193 M | 0.49 s |
| | W-2102 @ 2.9 GHz | AVX-512 | 1,349 M | 0.32 s |
| | W-2102 (2 cores) | AVX-512 | 1,388 M | 0.20 s |

# Emoji Search: icgrep

## Example: Search for Smileys

Regex: `\p{name=/SMIL(E|ING)/}`

- Data source: 620 MB Wikibooks document set (15 languages)

# Emoji Search: icgrep

## Example: Search for Smileys

Regex: `\p{name=/SMIL(E|ING)/}`

- Data source: 620 MB Wikibooks document set (15 languages)

## Performance Results

| Program | Processor | SIMD | Instructions | Time |
|---|---|---|---|---|
| icgrep | i7-3770 @ 3.4 GHz | SSE2 | 4,610 M | 0.55 s |
| | i3-5010U @ 2.1 GHz | AVX2 | 2,687 M | 0.59 s |
| | W-2102 @ 2.9 GHz | AVX-512 | 1,795 M | 0.38 s |
| | W-2102 (2 cores) | AVX-512 | 1,820 M | 0.23 s |

# Outline


1. Introduction to SIMD/AVX-512
2. Parabix: Scalable High-Performance Unicode
3. Bitwise Data Parallel Regular Expression Matching
4. Programming Framework: Kernels + Stream Sets = Programs
5. `icgrep` Architecture
6. Scalable Performance Results
7. Conclusion

# Final Remarks

## AVX-512 Scalability

- Instruction count drops dramatically, CPU time drops significantly.
- AVX-512 detection and code generation is automatic for Parabix applications.
- Performance improvement is automatic with significant reduction in both instruction count and execution time in most cases.
    - Improvement of core libraries is an ongoing area of work.

# Final Remarks

## AVX-512 Scalability

- Instruction count drops dramatically, CPU time drops significantly.
- AVX-512 detection and code generation is automatic for Parabix applications.
- Performance improvement is automatic with significant reduction in both instruction count and execution time in most cases.
  - Improvement of core libraries is an ongoing area of work.

## Parabix Platform

- Kernel + Stream Set model is effective for Parabix program design.
- Kernel library includes transposition and inverse transposition, stream filtering and stream expansion.
- Character class and Unicode property compilers.
- Pipeline compiler supports segmented multicore parallelism automatically.