Parabix Technology with icgrep

Robert D. Cameron

School of Computing Science
Simon Fraser University

May 21, 2021
Outline

1. Parabix: Scalable High-Performance Unicode
2. Bitwise Data Parallel Regular Expression Matching
3. Programming Framework: Kernels + Stream Sets = Programs
4. icgrep Architecture
5. Scalable Performance Results
6. 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.
- Parabix can also scale to use multiple cores, even on a single data stream.
- No changes to application programs required!
icgrep 1.8

- Full-featured grep implementation using Parabix algorithms.
- Posix REs: Basic or Extended
  - All features except backreferences.
- Perl-compatible REs (PCRE)
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. Parabix: Scalable High-Performance Unicode
2. Bitwise Data Parallel Regular Expression Matching
3. Programming Framework: Kernels + Stream Sets = Programs
4. icgrep Architecture
5. Scalable Performance Results
6. 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.
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 $a$.

\[
\begin{align*}
\text{input data} & \quad a453z--b3z--az--a12949z--ca22z7-- \\
M_1 & \quad .1............1...1.............1.......
\end{align*}
\]
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

Consider $M_2 = \text{MatchStar}(M_1; C)$

We use addition to scan each marker through the class. Bits that change represent matches.

We also have matches at start positions in input data.

$$a453z--b3z--az--a12949z--ca22z7--$$

$$M_1.1...........1...1.........1.....$$

$$C = [0-9].111....1........11111.....11.1..$$

$$T_0 = M_1^C.1...............1.........1.....$$

$$T_1 = T_0 + C....1...1.............1......11..$$

$$T_2 = T_1 + C.1111............111111....111...$$

$$M_2 = T_2 \_ M_1.1111........1...111111....111...$$
Matching Character Class Repetitions with MatchStar

- $\text{MatchStar}(M, C) = (((M \land C) + C) \oplus C) \lor M$
Matching Character Class Repetitions with MatchStar

- \( \text{MatchStar}(M, C) = ((M \land C) + C) \oplus C \lor M \)
- Consider \( M_2 = \text{MatchStar}(M_1, C) \)

### Example

**Input Data**

\[ M_1 \]

\[ C = [0-9] \]

<table>
<thead>
<tr>
<th>input data</th>
<th>a453z--b3z--az--a12949z--ca22z7--</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M_1 )</td>
<td>.1................1...1...........1.....</td>
</tr>
<tr>
<td>( C = [0-9] )</td>
<td>.111.....1........11111......11.1..</td>
</tr>
</tbody>
</table>
Matching Character Class Repetitions with MatchStar

- MatchStar\((M, C) = (((M \land C) + C) \oplus C) \lor M\)
- Consider \(M_2 = \text{MatchStar}(M_1, C)\)
- Use addition to scan each marker through the class.

<table>
<thead>
<tr>
<th>input data</th>
<th>a453z--b3z--az--a12949z--ca22z7--</th>
</tr>
</thead>
<tbody>
<tr>
<td>(M_1)</td>
<td>.1.........1..111111....11111...</td>
</tr>
<tr>
<td>(C = [0-9])</td>
<td>.111....1........111111.....11.1..</td>
</tr>
<tr>
<td>(T_0 = M_1 \land C)</td>
<td>.111111....111111....11111...</td>
</tr>
<tr>
<td>(T_1 = T_0 + C)</td>
<td>....1...111111....111111.....111..</td>
</tr>
</tbody>
</table>

Rob Cameron (SFU)
Parabix/icgrep
May 21, 2021 11 / 29
Matching Character Class Repetitions with MatchStar

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

\[
\text{input data} \quad a453z--b3z--az--a12949z--ca22z7--
\]
\[
M_1 = .1................1...1........1.....
\]
\[
C = [0-9] \quad .111....1........11111.....11.1..
\]
\[
T_0 = M_1 \land C \quad .1................1........1.....
\]
\[
T_1 = T_0 + C \quad .....1...1..............1.........1......11..
\]
\[
T_2 = T_1 \oplus C \quad .1111...........111111.....111...
\]
Matching Character Class Repetitions with MatchStar

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

<table>
<thead>
<tr>
<th>input data</th>
<th>a453z--b3z--az--a12949z--ca22z7--</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M_1 )</td>
<td>.1................1...1............1.</td>
</tr>
<tr>
<td>( C = [0-9] )</td>
<td>.111.....1........111111......11.1...</td>
</tr>
<tr>
<td>( T_0 = M_1 \land C )</td>
<td>.1............1....1.............1.....</td>
</tr>
<tr>
<td>( T_1 = T_0 + C )</td>
<td>....1...1.............1......11......</td>
</tr>
<tr>
<td>( T_2 = T_1 \oplus C )</td>
<td>.1111.............111111.....111....</td>
</tr>
<tr>
<td>( M_2 = T_2 \lor M_1 )</td>
<td>.1111.............1...111111.....111...</td>
</tr>
</tbody>
</table>
Matching Equations

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

\[
\begin{align*}
\text{Match}(m, C) &= \text{Advance}(\text{CharClass}(C) \land m) \\
\text{Match}(m, RS) &= \text{Match}(\text{Match}(m, R), S) \\
\text{Match}(m, R|S) &= \text{Match}(m, R) \lor \text{Match}(m, S) \\
\text{Match}(m, C*) &= \text{MatchStar}(m, \text{CharClass}(C)) \\
\text{Match}(m, R*) &= m \lor \text{Match}(\text{Match}(m, R), R*) \\
\text{Advance}(m) &= m + m \\
\text{MatchStar}(m, C) &= (((m \land C) + C) \oplus C) \lor m
\end{align*}
\]

The recursive equation is implemented with a while loop.
1. Parabix: Scalable High-Performance Unicode

2. Bitwise Data Parallel Regular Expression Matching

3. Programming Framework: Kernels + Stream Sets = Programs

4. icgrep Architecture

5. Scalable Performance Results

6. Conclusion
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).
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 of 8 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 × i1: the 8 basis bit streams.
- Output: 3 × 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 × i1: the 8 basis bit streams.
- Output: 3 × 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 × i1: character class streams
- Output: 1 × i1: a bit stream of matches found.
- Dynamically generated by the Parabix Regular Expression compiler.
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, ...
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, ...

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. Parabix: Scalable High-Performance Unicode
2. Bitwise Data Parallel Regular Expression Matching
3. Programming Framework: Kernels + Stream Sets = Programs
4. icgrep Architecture
5. Scalable Performance Results
6. Conclusion
Parabix Compilation Architecture: icgrep

RegEx

RegEx Parser

RegEx Transformations

RegEx Compiler

Pablo Transformations

Pablo Compiler

Pipeline Compiler

LLVM Compiler

Parabix Driver

SIMD Detection

Kernel Libraries

SIMD Libraries

Object Cache

Dynamically-Generated Match Function
1. Parabix: Scalable High-Performance Unicode
2. Bitwise Data Parallel Regular Expression Matching
3. Programming Framework: Kernels + Stream Sets = Programs
4. icgrep Architecture
5. Scalable Performance Results
6. 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

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

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

<table>
<thead>
<tr>
<th>Program</th>
<th>Processor</th>
<th>SIMD</th>
<th>Instructions</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>grep -P</td>
<td>i7-3770 @ 3.4 GHz</td>
<td>SSE2</td>
<td>2,191,635 M</td>
<td>232.3 s</td>
</tr>
<tr>
<td></td>
<td>i3-5010U @ 2.1 GHz</td>
<td>AVX2</td>
<td>2,191,744 M</td>
<td>348.0 s</td>
</tr>
<tr>
<td></td>
<td>W-2102 @ 2.9 GHz</td>
<td>AVX-512</td>
<td>2,191,552 M</td>
<td>220.8 s</td>
</tr>
<tr>
<td>icgrep</td>
<td>i7-3770 @ 3.4 GHz</td>
<td>SSE2</td>
<td>6,678 M</td>
<td>0.85 s</td>
</tr>
<tr>
<td></td>
<td>i3-5010U @ 2.1 GHz</td>
<td>AVX2</td>
<td>3,683 M</td>
<td>0.84 s</td>
</tr>
<tr>
<td></td>
<td>W-2102 @ 2.9 GHz</td>
<td>AVX-512</td>
<td>2,174 M</td>
<td>0.44 s</td>
</tr>
<tr>
<td></td>
<td>W-2102 (2 cores)</td>
<td>AVX-512</td>
<td>2,206 M</td>
<td>0.25 s</td>
</tr>
</tbody>
</table>
Large Bounded Repetitions

Example: Lines $\geq 400$ Characters

Regex: .{400}

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

Example: Lines $\geq 400$ Characters

Regex: `.\{400\}`
- Data source: 620 MB Wikibooks document set (15 languages)
- icgrep has $\log_2$ algorithm.

Performance Results

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

Example: IP address regex

```regex
(25[0-5]|2[0-4][0-9]|1?[0-9][0-9]?)
(\.(25[0-5]|2[0-4][0-9]|1?[0-9][0-9]?)\{3\})
```

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

**Example: IP address regex**

```plaintext
(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

<table>
<thead>
<tr>
<th>Program</th>
<th>Processor</th>
<th>SIMD</th>
<th>Instructions</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>grep -E</td>
<td>i7-3770 @ 3.4 GHz</td>
<td>SSE2</td>
<td>232,079 M</td>
<td>21.3 s</td>
</tr>
<tr>
<td></td>
<td>i3-5010U @ 2.1 GHz</td>
<td>AVX2</td>
<td>232,423 M</td>
<td>39.5 s</td>
</tr>
<tr>
<td></td>
<td>W-2102 @ 2.9 GHz</td>
<td>AVX-512</td>
<td>232,081 M</td>
<td>25.6 s</td>
</tr>
<tr>
<td>icgrep</td>
<td>i7-3770 @ 3.4 GHz</td>
<td>SSE2</td>
<td>3,720 M</td>
<td>0.49 s</td>
</tr>
<tr>
<td></td>
<td>i3-5010U @ 2.1 GHz</td>
<td>AVX2</td>
<td>2,193 M</td>
<td>0.49 s</td>
</tr>
<tr>
<td></td>
<td>W-2102 @ 2.9 GHz</td>
<td>AVX-512</td>
<td>1,349 M</td>
<td>0.32 s</td>
</tr>
<tr>
<td></td>
<td>W-2102 (2 cores)</td>
<td>AVX-512</td>
<td>1,388 M</td>
<td>0.20 s</td>
</tr>
</tbody>
</table>
Emoji Search: icgrep

Example: Search for Smileys

Regex: \p{name=/SMILEY/}

- 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

<table>
<thead>
<tr>
<th>Program</th>
<th>Processor</th>
<th>SIMD</th>
<th>Instructions</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>icgrep</td>
<td>i7-3770 @ 3.4 GHz</td>
<td>SSE2</td>
<td>4,610 M</td>
<td>0.55 s</td>
</tr>
<tr>
<td></td>
<td>i3-5010U @ 2.1 GHz</td>
<td>AVX2</td>
<td>2,687 M</td>
<td>0.59 s</td>
</tr>
<tr>
<td></td>
<td>W-2102 @ 2.9 GHz</td>
<td>AVX-512</td>
<td>1,795 M</td>
<td>0.38 s</td>
</tr>
<tr>
<td></td>
<td>W-2102 (2 cores)</td>
<td>AVX-512</td>
<td>1,820 M</td>
<td>0.23 s</td>
</tr>
</tbody>
</table>
Outline

1. Parabix: Scalable High-Performance Unicode

2. Bitwise Data Parallel Regular Expression Matching

3. Programming Framework: Kernels + Stream Sets = Programs

4. icgrep Architecture

5. Scalable Performance Results

6. 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.