Bitwise Data Parallelism with LLVM: The ICgrep Case Study

Robert D. Cameron Nigel Medforth Dan Lin Dale Denis
William N. Sumner

School of Computing Science
Simon Fraser University

November 18, 2015

Overview

The Parabix Regular Expression Project

- Goal: building a modern, ultra-fast, Unicode-compliant, regular expression engine and application framework.
- New algorithmic approach: bitwise data parallel matching.
- Performance target: Gigabyte/second regexp processing.
- Status: icgrep 1.0 is a full Unicode Level 1 grep replacement.
- Current work: Unicode Level 2 Regular Expression Support
- Longer-term: high-level application framework.

Performance Matters - Big Text

Text Analytics

- Text Analytics applications: very large textual data sets.
- Gigabyte to Terabyte scale.
- Regular expressions are often used in these applications.

Performance Matters - Big Text

Text Analytics

- Text Analytics applications: very large textual data sets.
- Gigabyte to Terabyte scale.
- Regular expressions are often used in these applications.

IBM System T Study

- Five customer workloads studied
- Four of five workloads dominated by regexp processing.
- 60%-75% of total execution time.
- Polig *et al.* Giving text analytics a boost. *IEEE Micro*, **34**, 2014.

Much Better Performance is Possible

Unicode regular expression matching can be ultra-fast and robust.

Example: email regex with Unicode properties

- `([^\p{Z}<]+@[\p{L}\p{M}\p{N}.-]+\.`
  `(\p{L}\p{M}*){2,6})(>|\p{Z}|$)`
- Compare time in CPU cycles per byte for 4 grep programs.
- Data source: 620 MB Wikibooks document set (15 languages)
- Processor: Intel Core i7-2600 CPU @ 3.40GHz

Much Better Performance is Possible

Unicode regular expression matching can be ultra-fast and robust.

Example: email regex with Unicode properties

- `([^\p{Z}<]+@[\p{L}\p{M}\p{N}.-]+\.`
  `(\p{L}\p{M}*){2,6})(>|\p{Z}|$)`
- Compare time in CPU cycles per byte for 4 grep programs.
- Data source: 620 MB Wikibooks document set (15 languages)
- Processor: Intel Core i7-2600 CPU @ 3.40GHz

Performance Results

Program	Elapsed sec	Cycles/Byte	Megabytes/sec
ugrep561	180.7	1024.0	3.60
pcre2grep	144.0	815.5	4.51
grep210 -P	87.3	498.2	7.45
icgrep4750	1.06	6.05	613.0

Bitwise Data Parallel Regular Expression Matching

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. Bit stream j consists of bit j of each byte.

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. Bit stream j consists of bit j of each byte.

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. Bit stream j consists of bit j of each byte.

input data `a45a...` (`01100001 00110100 00110101 01100001...`)

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. Bit stream j consists of bit j of each byte.

```
input data   a45a... (01100001 00110100 00110101 01100001...)
  bit0       0000...
```

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. Bit stream j consists of bit j of each byte.

```
input data   a45a... (01100001 00110100 00110101 01100001...)
   bit0      0000...
   bit1      1001...
```

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. Bit stream j consists of bit j of each byte.

input data	a45a... (01100001 00110100 00110101 01100001...)
*bit*0	0000...
*bit*1	1001...
*bit*2	1111...
*bit*3	0110...
*bit*4	0000...
*bit*5	0110...
*bit*6	0000...
*bit*7	1011...

Beyond Byte-At-A-Time

- 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.

Character Class Formation

- Combining 8 bits of a code unit gives a character class stream.
- CCC(`cc_a = [a]`)
- ```
  temp1 = (bit1 &~ bit0)
  temp2 = (bit2 &~ bit3)
  temp3 = (temp1 & temp2)
  temp4 = (bit4 | bit5)
  temp5 = (bit7 &~ bit6)
  temp6 = (temp5 &~ temp4)
  cc_a = (temp3 & temp6)
  ```

input data	a45a...
cc_a	1001...
  ```

Character Class Ranges

- Ranges of characters are often very simple to compute.
- CCC(`cc_0_9 = [0-9]`)
- ```
  temp7 = (bit0 | bit1)
  temp8 = (bit2 & bit3)
  temp9 = (temp8 &~ temp7)
  temp10 = (bit5 | bit6)
  temp11 = (bit4 & temp10)
  cc_0_9 = (temp9 &~ temp11)
  ```
  ```

Character Class Common Subexpressions

- Multiple definitions use common subexpressions.
- CCC(`cc_z9 = [z9]`)
- ```
  temp12 = (bit4 &~ bit5)
  temp13 = (temp12 & temp5)
  temp14 = (temp9 & temp13)
  temp15 = (temp1 & temp8)
  temp16 = (bit6 &~ bit7)
  temp17 = (temp12 & temp16)
  temp18 = (temp15 & temp17)
  cc_z9 = (temp14 | temp18)
  ```
  ```

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.
- 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--
   M1        .1..........1...1.........1.....
   M2        .1111.......1...111111....111...
   M3        .....1.......1.....1.11......1..
```

Matching Character Class Repetitions with MatchStar

Matching Character Class Repetitions with MatchStar

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

Matching Character Class Repetitions with MatchStar

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

input data	a453z--b3z--az--a12949z--ca22z7--
M_1	.1.........1...1........1....
$C = [0\text{-}9]$	.111....1.......11111....11.1..

Matching Character Class Repetitions with MatchStar

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

input data	a453z--b3z--az--a12949z--ca22z7--
M_1	.1.........1...1........1....
$C = [0\text{-}9]$	.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 $M_2 = \text{MatchStar}(M_1, C)$
- 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....
$C = [0\text{-}9]$	.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 $M_2 = \text{MatchStar}(M_1, C)$
- Use addition to scan each marker through the class.
- Bits that change represent matches.
- We also have matches at start positions in M_1.

input data	a453z--b3z--az--a12949z--ca22z7--
M_1	.1.........1...1........1....
$C = [0\text{-}9]$	.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.

UTF-8 Regular Expression Matching

UTF-8 Byte Classification and Validation

$$
\begin{aligned}
\text{ASCII} &= \text{CharClass}(\texttt{[\textbackslash x00-\textbackslash x7F]}) \\
\text{Prefix} &= \text{CharClass}(\texttt{[\textbackslash xC2-\textbackslash F4]}) \\
\text{Pfx3or4} &= \text{CharClass}(\texttt{[\textbackslash xE0-\textbackslash xF4]}) \\
\text{Prefix4} &= \text{CharClass}(\texttt{[\textbackslash xF0-\textbackslash xF4]}) \\
\text{Suffix} &= \text{CharClass}(\texttt{[\textbackslash x80-\textbackslash xBF]}) \\
\text{Scope} &= \text{Advance}(\text{Prefix}) \vee \text{Advance}(\text{Pfx3or4}, 2) \vee \text{Advance}(\text{Prefix4}, 3) \\
\text{Mismatch} &= \text{Scope} \oplus \text{Suffix} \\
\text{Initial} &= \text{ASCII} \vee \text{Prefix} \\
\text{NonFinal} &= \text{Prefix} \vee \text{Advance}(\text{Pfx3or4}) \vee \text{Advance}(\text{Prefix4}, 2)
\end{aligned}
$$

UTF-8 Matching Operations

Advance and Matchstar can both be adapted for UTF-8 matching. Let U8Class(U) be the stream marking the *final* UTF-8 byte position of occurrences of members of the class.

$$\begin{aligned} \text{Match}(m, U) &= \text{Advance}(\text{ScanThru}(m, \text{NonFinal}) \wedge \text{U8Class}(U)) \\ \text{Match}(m, U*) &= \text{MatchStar}(m, \text{U8Class}(U) \vee \text{NonFinal}) \wedge \text{Initial} \\ \text{ScanThru}(m, C) &= (m + C) \wedge \neg C \end{aligned}$$

Simple UTF-8 Matching Operations

- Search for 你好 in UTF-8 text.
- Let ni3 represent the three byte sequence for 你.
- Let hao represent the three byte sequence for 好。
- Let men represent the three byte sequence for 们。

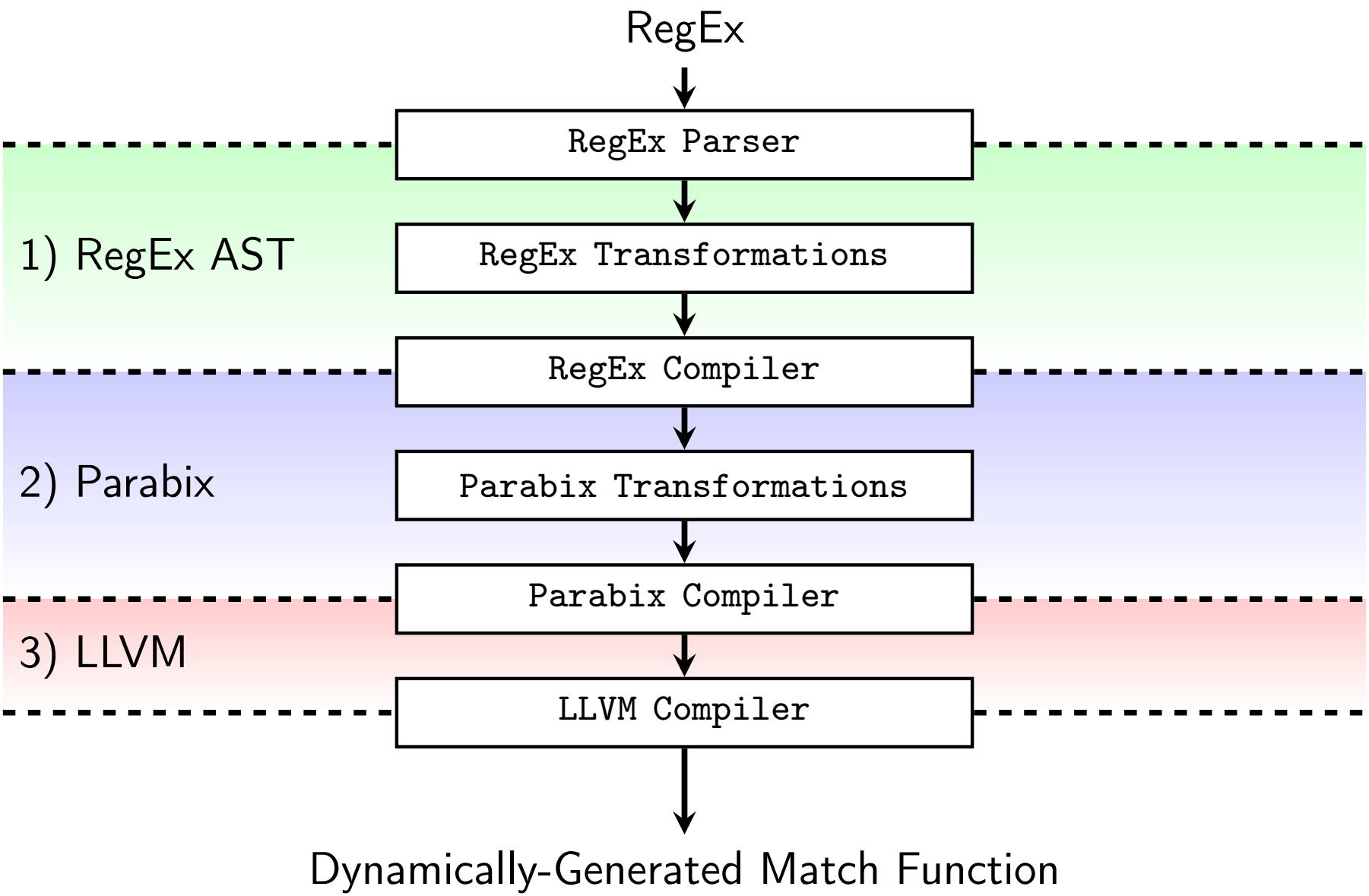
input data	`ni3hao(Hello),ni3men(You),`
CC_{ni3}	`..1.........1.....`
CC_{hao}	`.....1.............`
$m_0 = \text{Initial}$	`1..1..111111111..1..111111`
NonFinal	`11.11.....11.11.....`
$t_1 = \text{ScanThru}(m_0, \text{NonFinal})$	`..1..111111111..1..1111111`
$m_1 = \text{Advance}(t_1 \wedge CC_{ni3})$	`...1.........1.....`
$t_2 = \text{ScanThru}(m_1, \text{NonFinal})$	`.....1.........1.....`
$m_2 = \text{Advance}(t_2 \wedge CC_{hao})$	`......1.............`

If Hierarchies

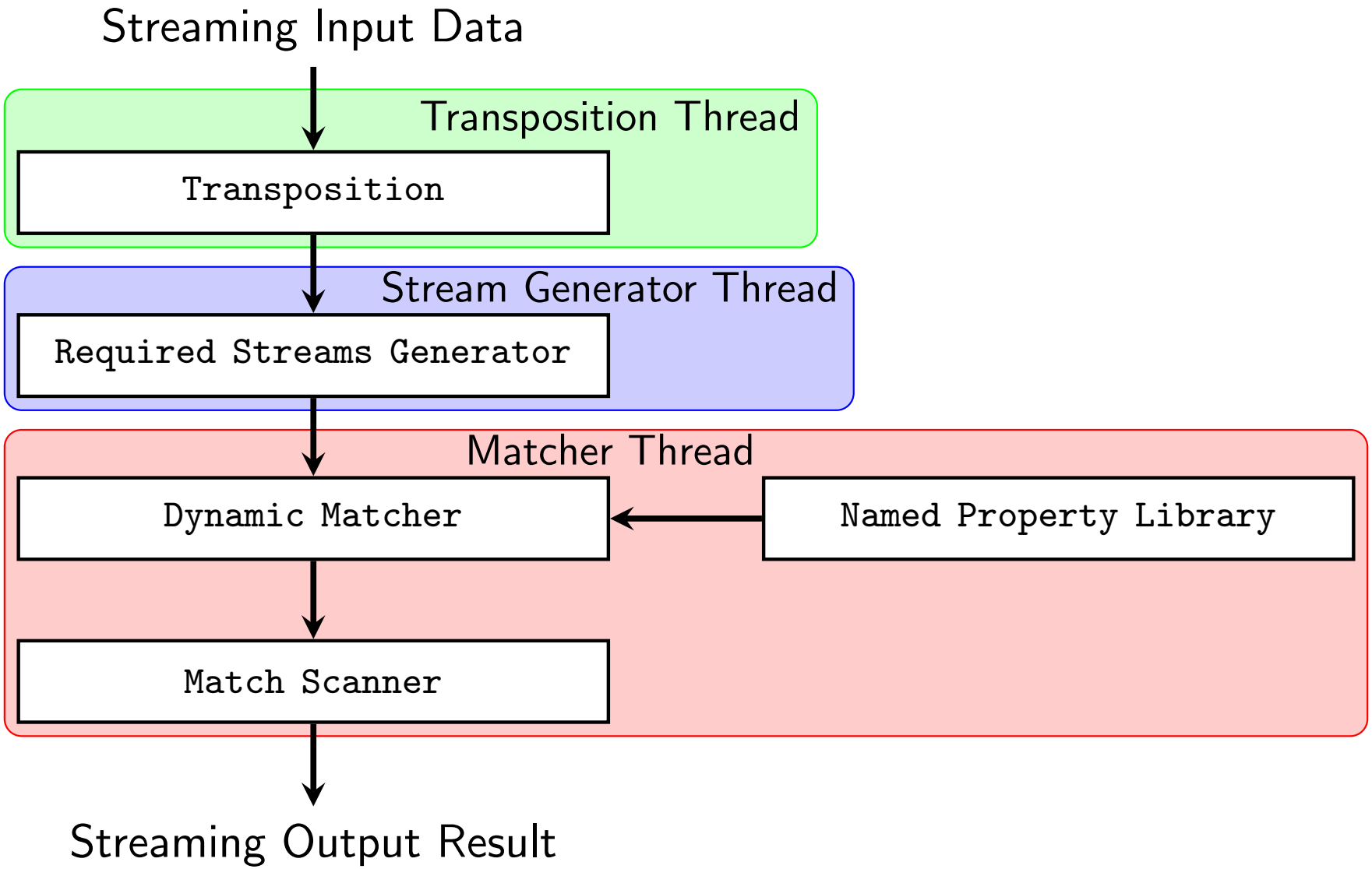
- Unicode property classes may contain many thousands of codepoints.
- Evaluating all required equations may be very expensive.
- However, any 128-byte segment of input will involve only a few codepoint ranges.
- Evaluation of property classes is embedded in if-hierarchies of successively finer ranges.
- This techique greatly reduces the number of equations evaluated for each property.

Architecture

Compilation Architecture



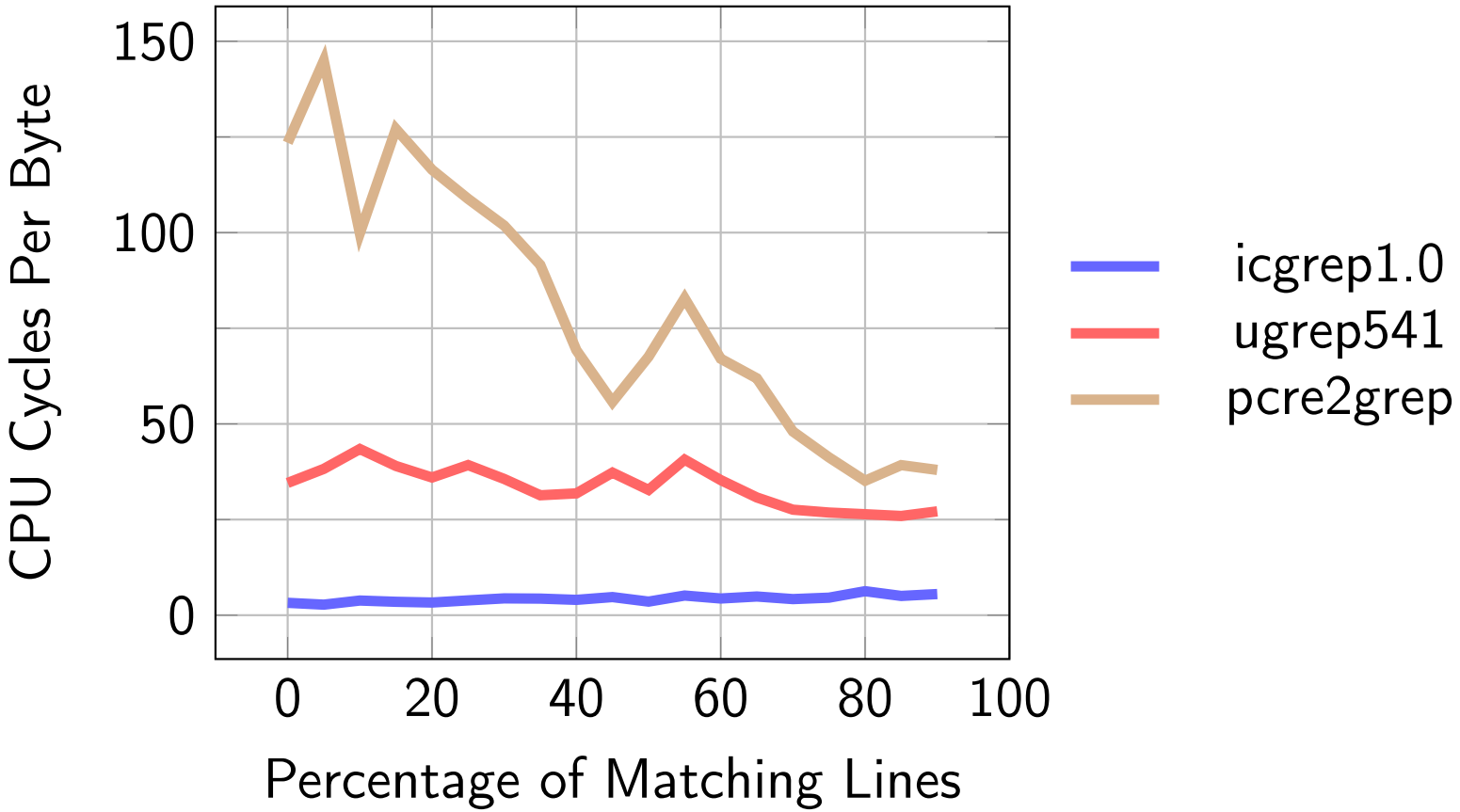
Execution Architecture



Performance Study Set-up

- Search for lines containing characters in computed sets.
- Set expressions combine general category and script properties.
- Set intersection, union, difference, negation.
- Cover all general categories and scripts.
- 246 test expressions in all.
- Evaluate icgrep 1.0 vs. ugrep541 and pcre2grep.
- Measure performance against a large collection of Wikimedia documents.

Performance Comparison



Performance Comparison (Multithread)

- Pipeline parallelism
- Up to 40% speedup by hiding transposition and required stream generation.
- Combining the AVX2 ISA with multithreading gives and average overall 61%
- Future work
  - Pipeline + data parallelism
  - AVX-512 ISA

Conclusion

Results

- Bitwise data parallelism is well-suited to Unicode regular expression matching requirements.
- Dramatic speed-up over sequential methods: often 10X or better.
- Performance improves as SIMD processor width increases.
- Further performance enhancement with multithreading: pipeline, task and/or coarse-grained data parallelism.

Conclusion

Results

- Bitwise data parallelism is well-suited to Unicode regular expression matching requirements.
- Dramatic speed-up over sequential methods: often 10X or better.
- Performance improves as SIMD processor width increases.
- Further performance enhancement with multithreading: pipeline, task and/or coarse-grained data parallelism.

icGrep 1.0

- Complete modern grep with full Unicode Level 1 support.
- Open source: `http://parabix.costar.sfu.ca/wiki/ICgrep`
- Many options for teaching and research.
- Download it, use it and give us your feedback!