Metal with Rust: High-Performance Regex Execution on Unified Memory GPUs

Bachelor Thesis Presentation

Marius De Kuthy Meurers

08.04.2025

School of Computation, Information and Technology
Technical University Munich

Introduction & Motivation

- **Problem:**
  - Increasing demand for fast processing of large amounts of texts.
  - Matching regexes on large numbers of lines is computationally expensive.
- **Opportunity:**
  - GPUs offer efficient parallel processing.
  - **BUT**: GPU–CPU memory transfer is expensive.
  - Unified Memory Architectures (UMA) potentially reduce transfer overhead.

$\Rightarrow$ Can we exploit Apple M1 Chips's UMA for regex matching performance gains?

Table of Contents

Regex Matching Process: Simple Regex Example

Input Lines

```
(614)-447-0327
(50-8) = 42
(202)-456-7041
08.04.2025
```

Regex Matching Engine

```
\([0-9]{3}\)-[0-9]{3}-[0-9]{4}
```

Match Results

```
true
false
true
false
```

Shift-And Algorithm

- Efficient regex matching using bitwise operations.

$\Rightarrow$ Encode regex transitions as bitmasks, realizing parallel state transitions.

Algorithm 1 Shift-And algorithm

Require: *mask_initial* ▷ bit vector for the start states
Require: *mask_final* ▷ bit vector for the final states
Require: *masks_char*[] ▷ bit vectors for the transitions
Require: *line* ▷ the input text

$state \leftarrow 0$

$n\_matches \leftarrow 0$

Shift-And Algorithm

- Efficient regex matching using bitwise operations.

$\Rightarrow$ Encode regex transitions as bitmasks, realizing parallel state transitions.

Algorithm 2 Shift-And algorithm

Require: $mask\_initial$ ▷ bit vector for the start states
Require: $mask\_final$ ▷ bit vector for the final states
Require: $masks\_char[]$ ▷ bit vectors for the transitions
Require: $line$ ▷ the input text
  $state \leftarrow 0$
  $n\_matches \leftarrow 0$
  for c in $line$ **do**
    $next \leftarrow (state \ll 1)$ OR $mask\_initial$ ▷ all possible next states
    $state \leftarrow next$ AND $masks\_char[c]$ ▷ mask for the current char

Shift-And Algorithm

- Efficient regex matching using bitwise operations.

$\Rightarrow$ Encode regex transitions as bitmasks, realizing parallel state transitions.

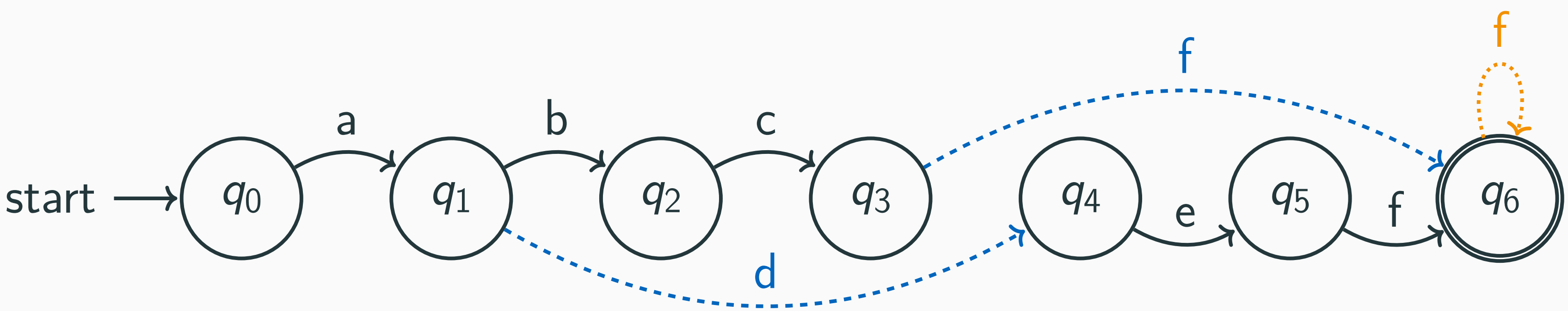
Algorithm 3 Shift-And algorithm

Require: *mask_initial* $\triangleright$ bit vector for the start states
Require: *mask_final* $\triangleright$ bit vector for the final states
Require: *masks_char*[] $\triangleright$ bit vectors for the transitions
Require: *line* $\triangleright$ the input text

$\quad state \leftarrow 0$
$\quad n\_matches \leftarrow 0$
$\quad$ **for** c in *line* **do**
$\quad\quad next \leftarrow (state \ll 1)$ OR $mask\_initial$ $\triangleright$ all possible next states
$\quad\quad state \leftarrow next$ AND $masks\_char[c]$ $\triangleright$ mask for the current char
$\quad\quad$ **if** $(state$ AND $mask\_final) \neq 0$ **then** $\triangleright$ check if any final state is reached
$\quad\quad\quad n\_matches \leftarrow n\_matches + 1$
$\quad$ **return** $n\_matches$

Automata with longer jumps

Finite automata for the regex `a(bc|de)f+` with different distance transitions.

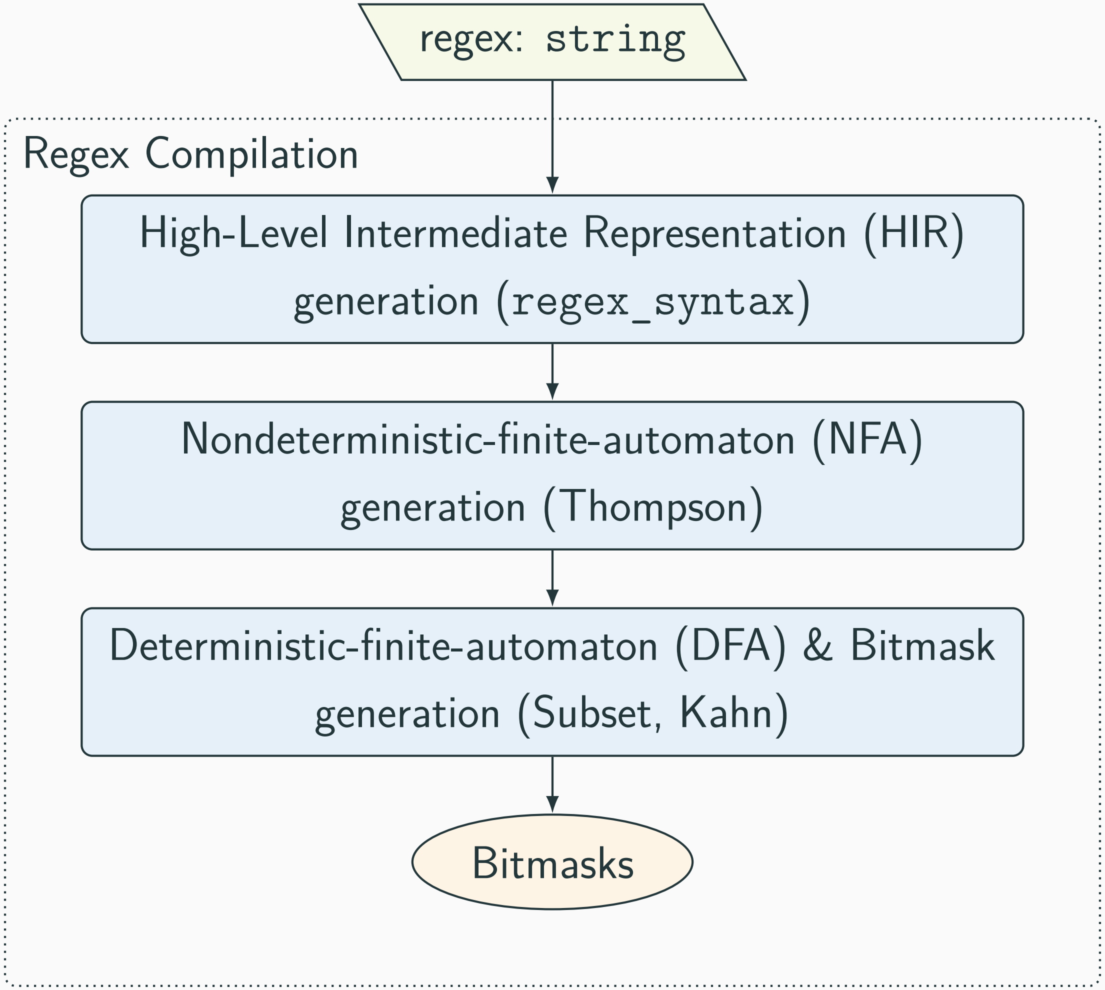


- Cannot be handled by the basic Shift-And Algorithm.

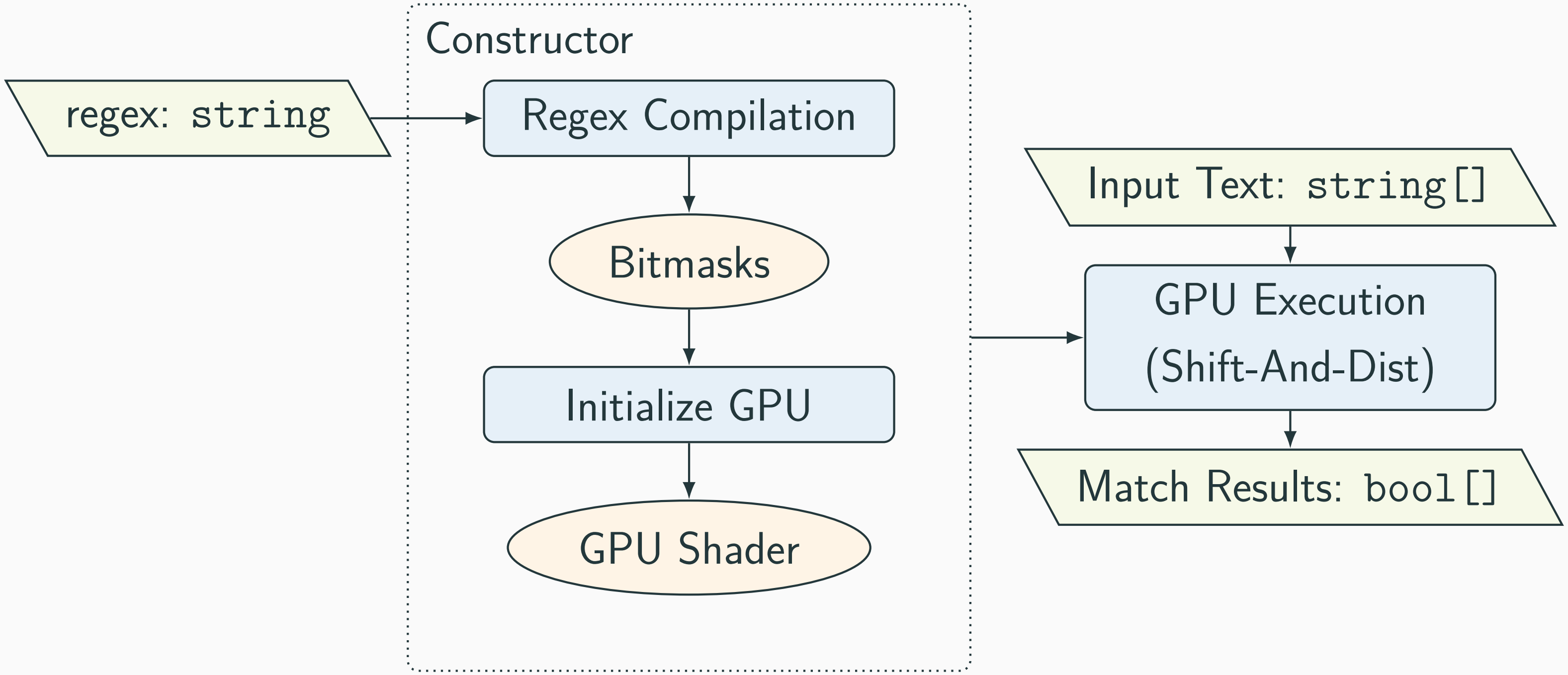
⇒ **Shift-And-Dist** algorithm: extension to support different distance jumps.

  - Limitation: No support for backwards transitions.

Implementation Overview

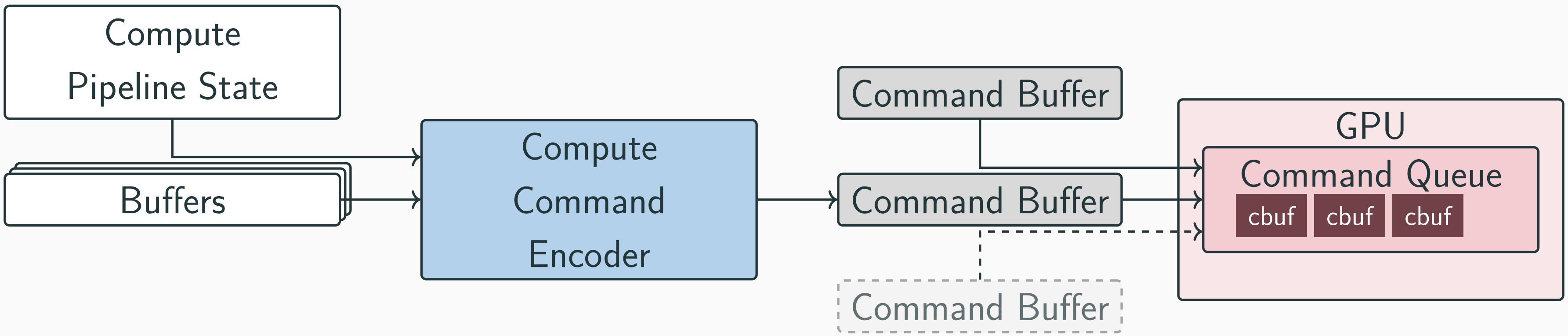


Implementation Overview



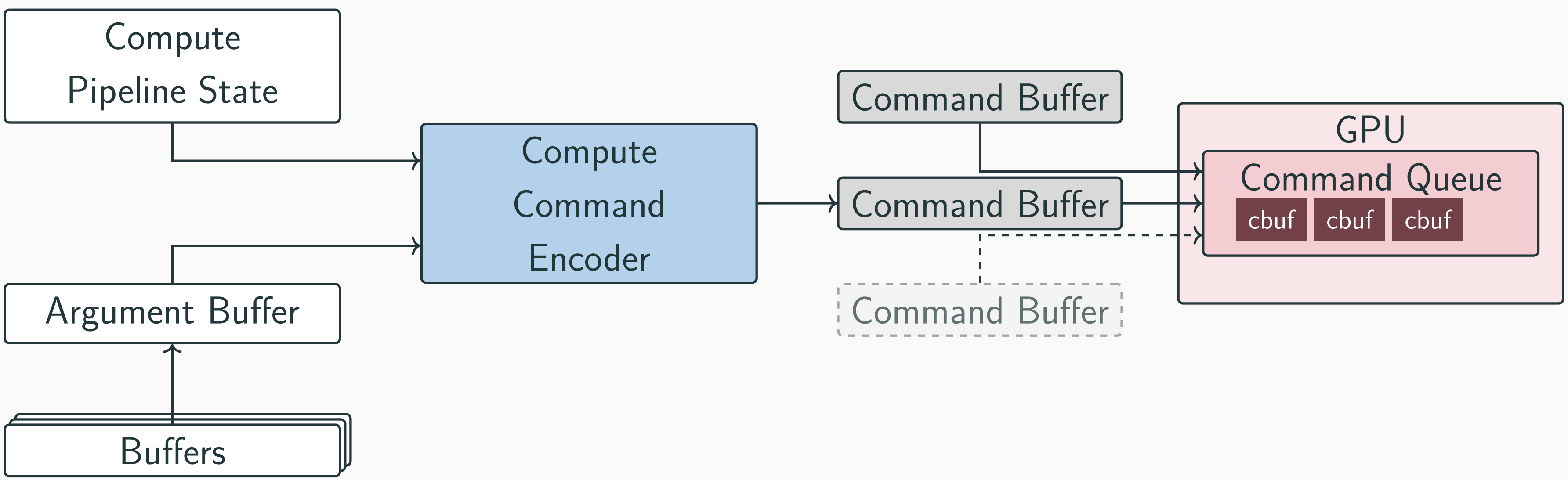
Metal Execution Model: Object relations and Metal pipeline

Approach 1: BasicStrategy



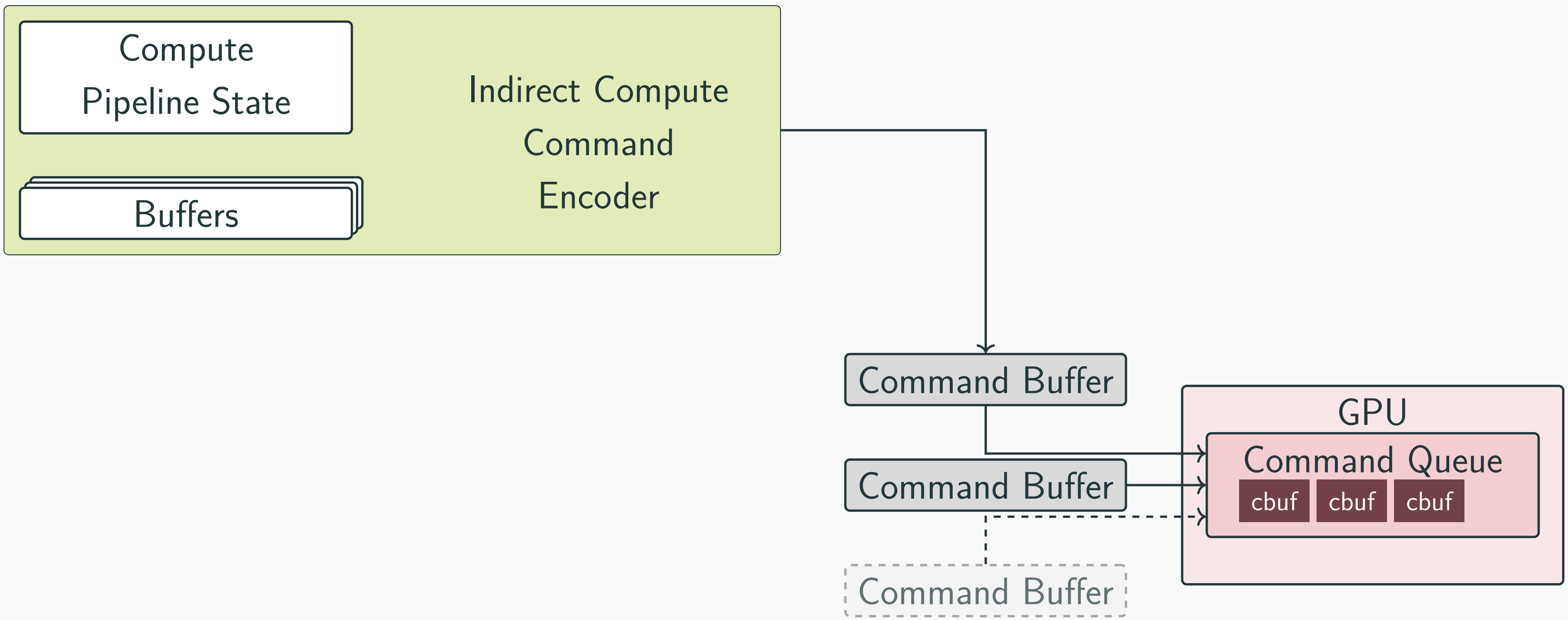
Metal Execution Model: Object relations and Metal pipeline

Approach 2: ArgumentBufferStrategy



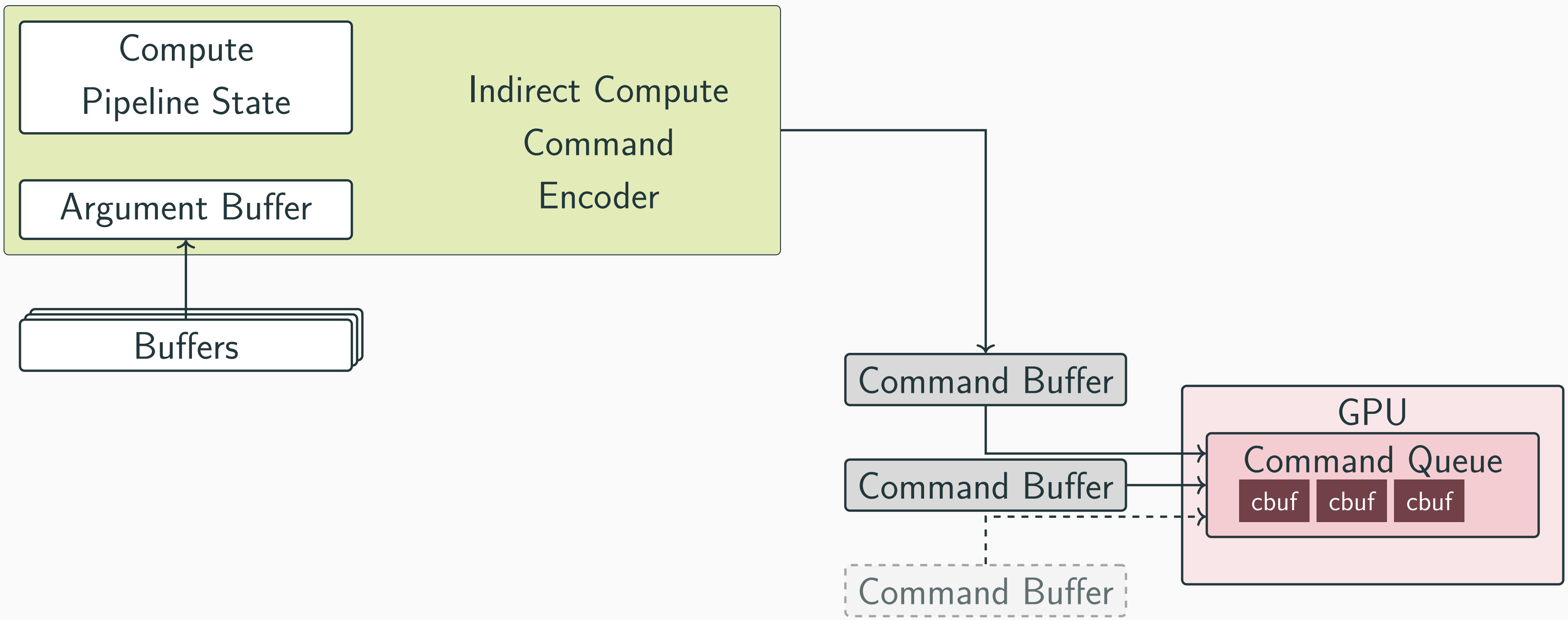
Metal Execution Model: Object relations and Metal pipeline

Approach 3: IndirectCommandBufferStrategy



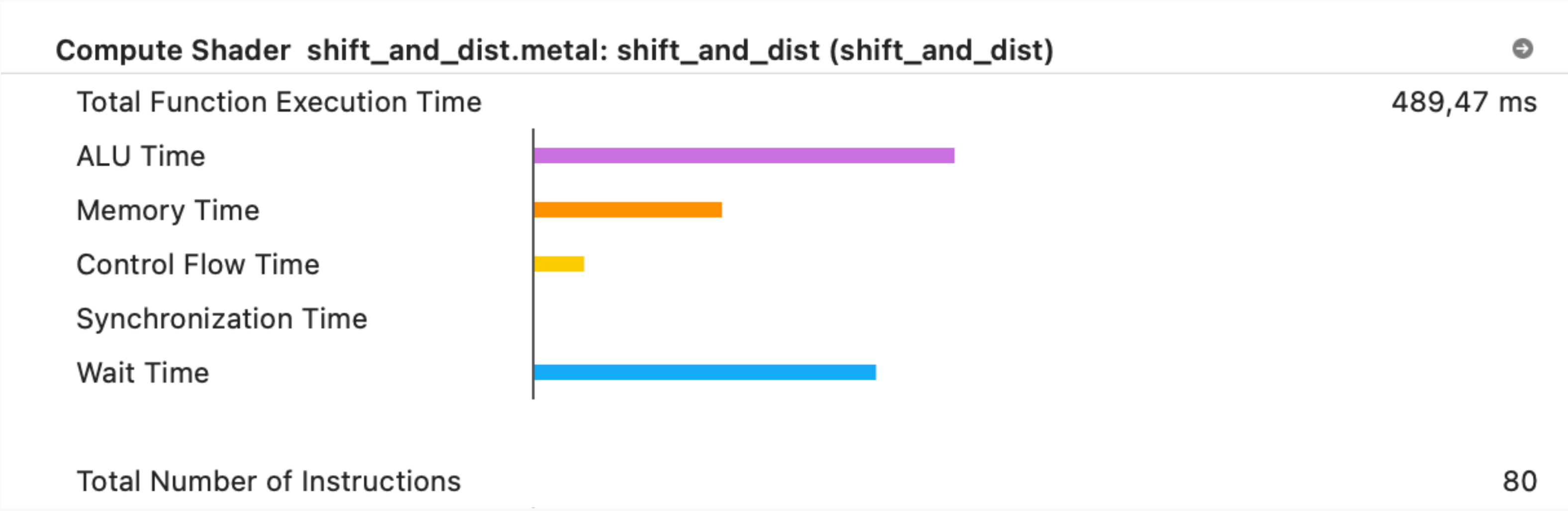
Metal Execution Model: Object relations and Metal pipeline

Approach 4: CombinedStrategy



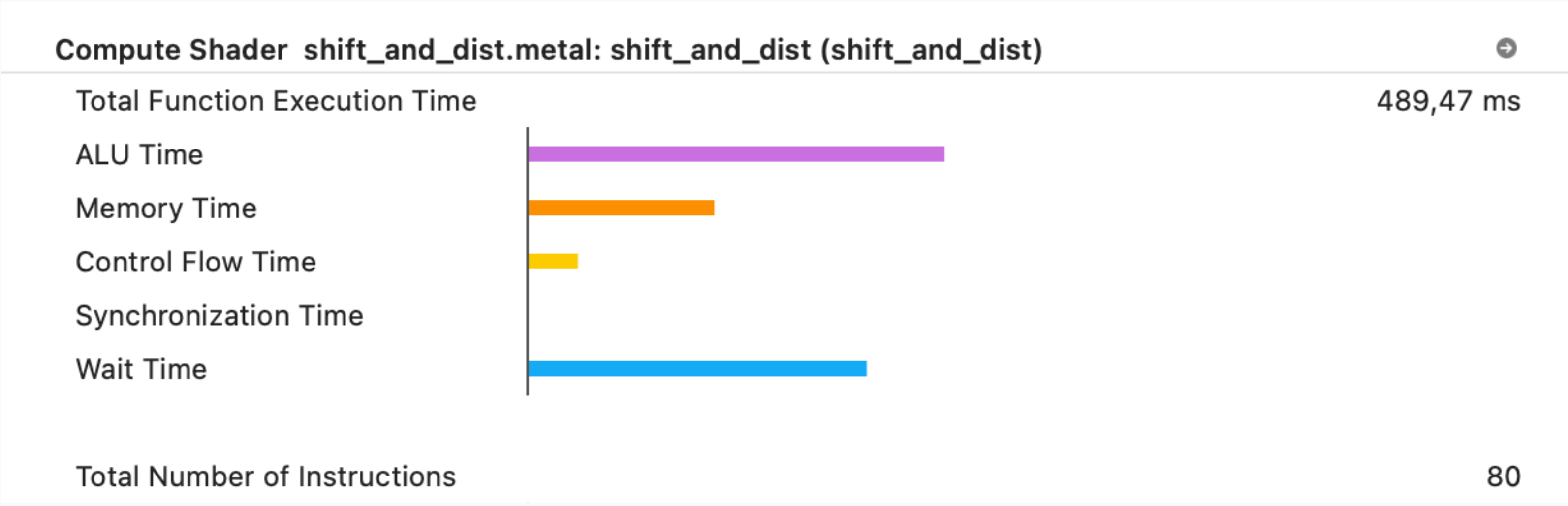
Evaluation: Generic or Just-in-Time Compiled Shader

Compiled Generic Shader

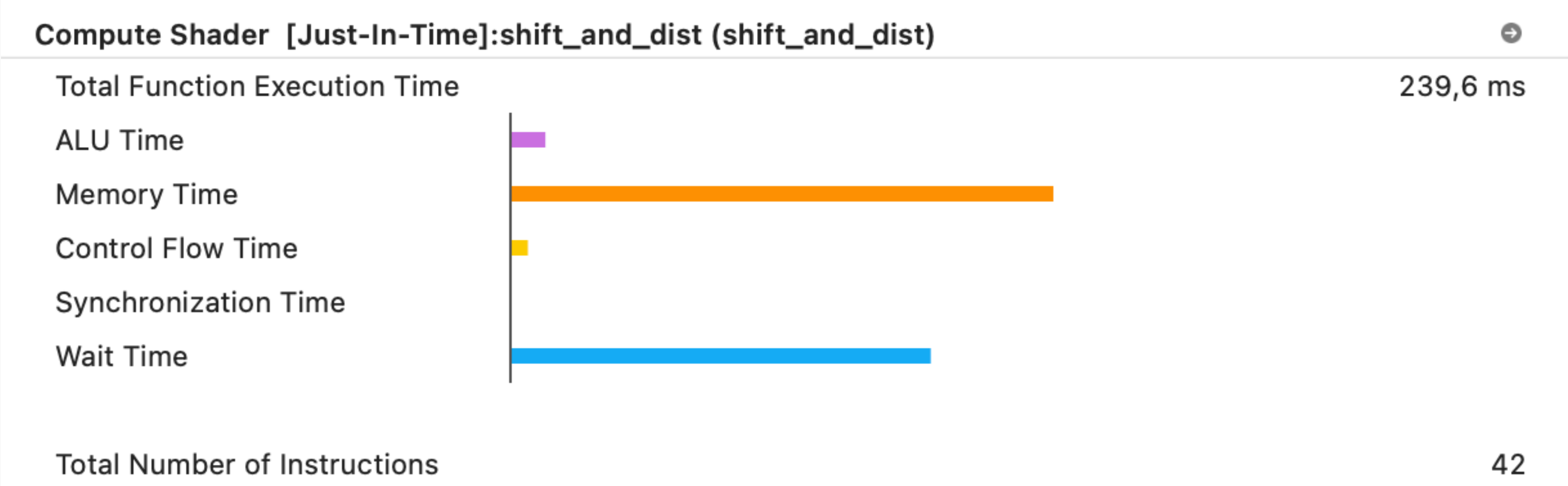


Evaluation: Generic or Just-in-Time Compiled Shader

Compiled Generic Shader



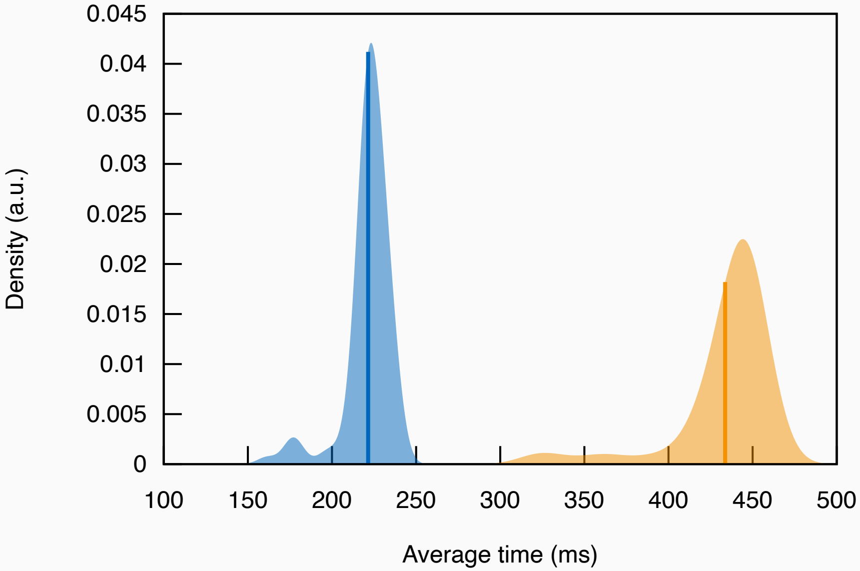
Just-in-time Optimized Shader



Evaluation: Just-in-Time Compilation Summary

JIT Compilation benefits on regex `a[^b]{62}b` processing 1 GiB:

- 48% faster execution time
- 42 instructions vs. 80 instructions
- 16.8B vs. 147.2B total instructions

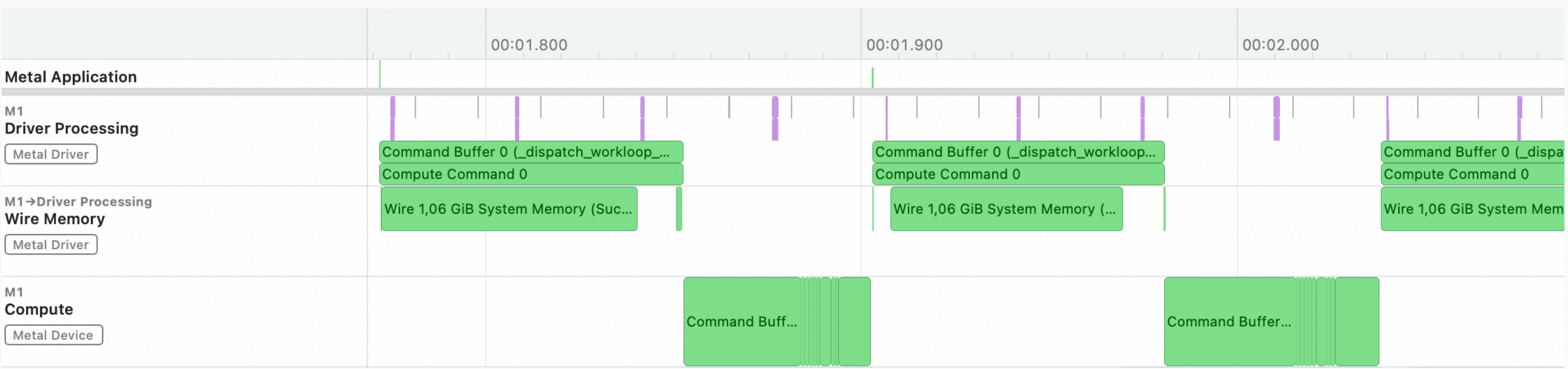


JIT(blue) vs. Generic(yellow)

⇒ JIT compilation significantly reduces execution time and instruction count.

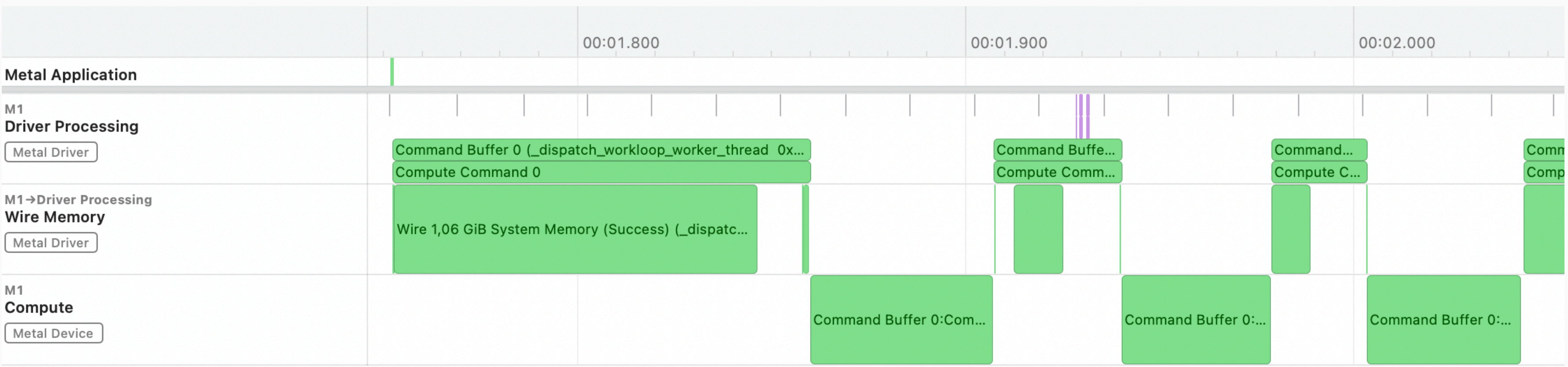
Evaluation: Unified Memory – Memory Wiring as Challenge

- **Apple's claim:** "Metal exposes UMA through shared resources that allow GPU and CPU to read/write the same memory"
- **Reality:**
  - initial GPU access requires "memory wiring" (∼90ms for 1.06 GiB)
  - 90% of driver processing time spent on this wiring memory
  - actual regex computation: only ∼50ms



Evaluation: Unified Memory – Memory Wiring

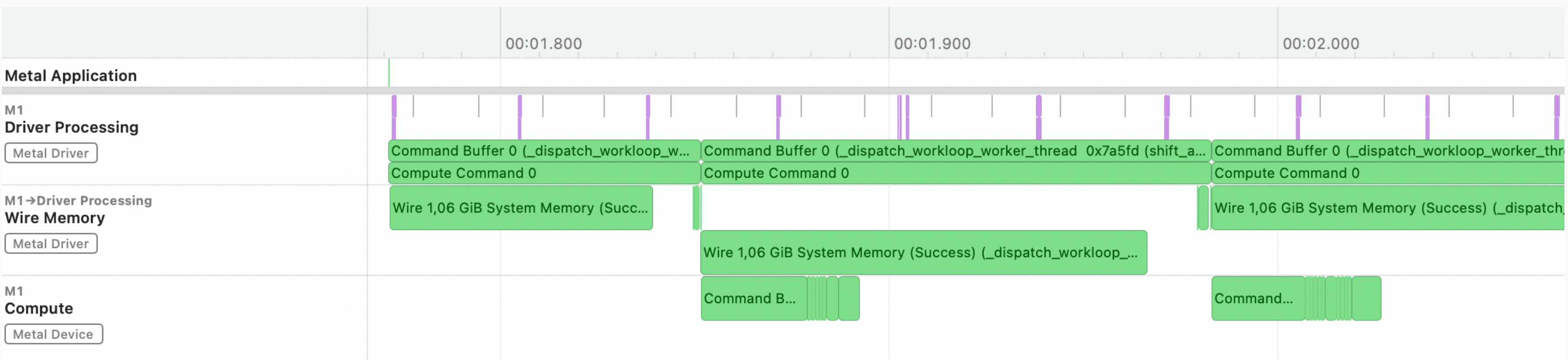
- **Caching:** Subsequent access to same memory cuts wiring time to <10ms



⇒ Further regex matching can take advantage of cached memory

Evaluation: Unified Memory – Async Processing

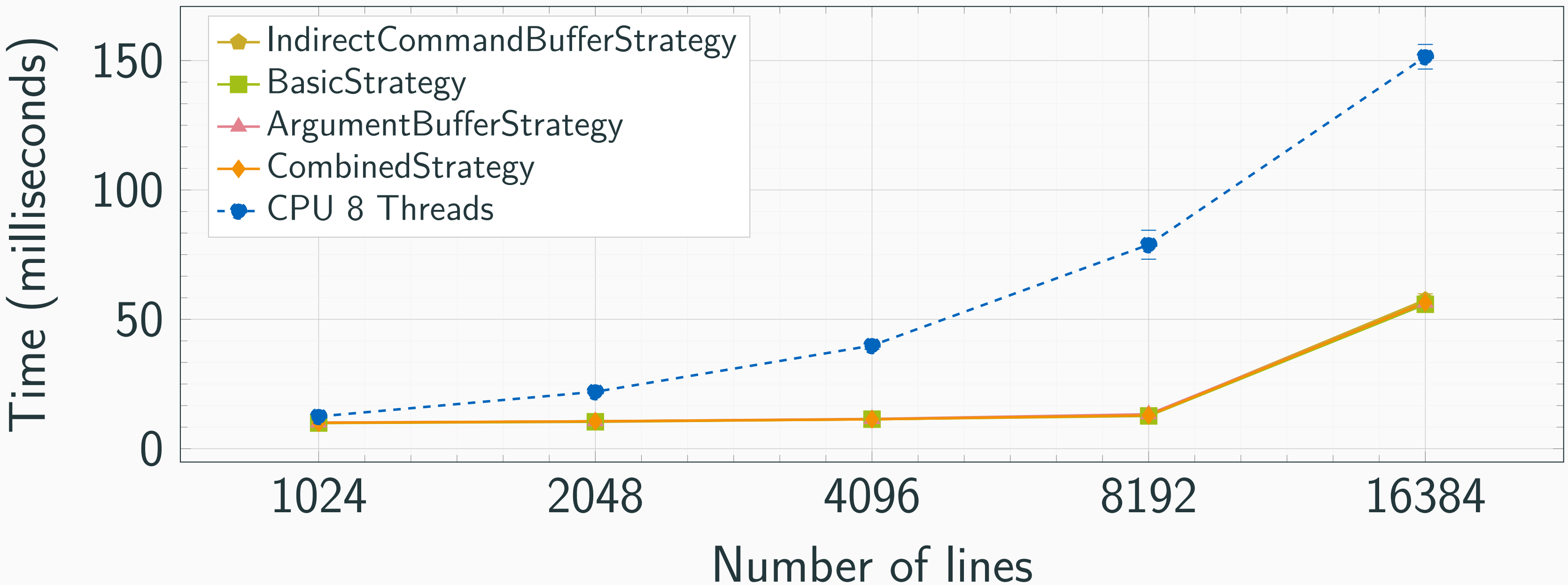
- **Challenge:** Memory wiring latency creates significant bottleneck
- **Solution:** Utilize Metal's completion handlers for asynchronous dispatch



⇒ Overlapping memory wiring with computation significantly reduces latency

Evaluation: Performance – Long Pattern

- **Regex:** `a[^b]{62}b` (utilizes all 64 states in uint64_t)



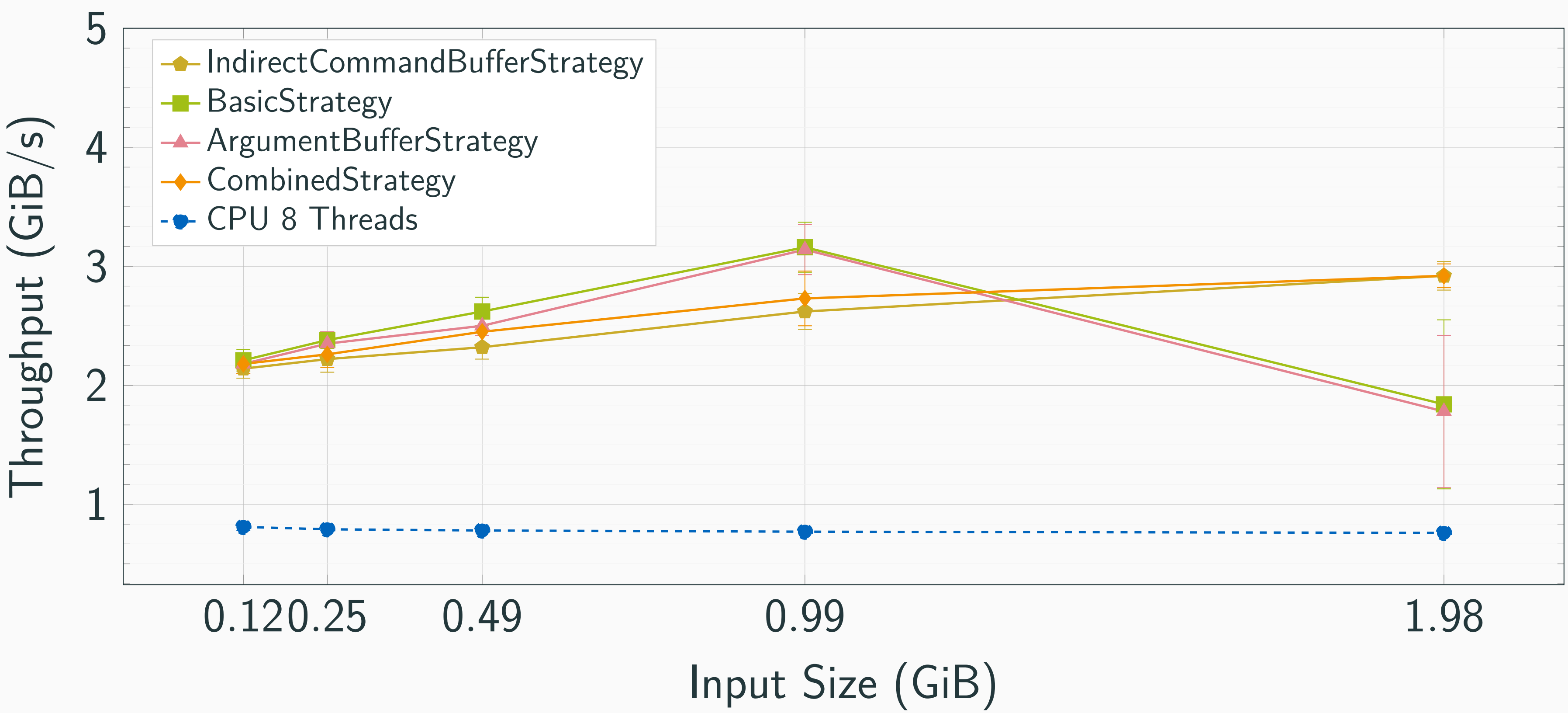
⇒ GPU: Up until 8192 lines constant execution time

⇒ GPU: All strategies performed similarly

⇒ CPU: Linear scaling with input size

Evaluation: Performance – Long Pattern

- **Regex:** `a[^b]{62}b`



⇒ Throughput grows with increasing input size and outperforms CPU

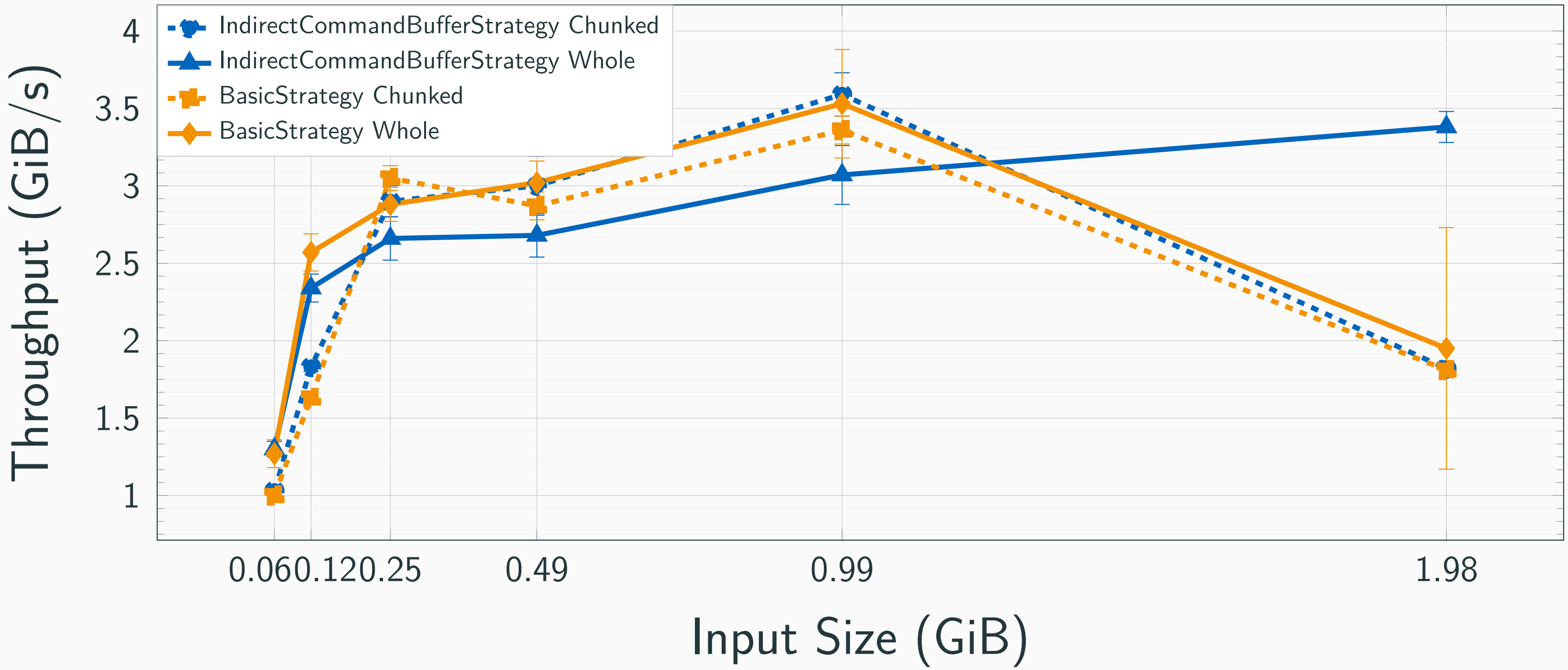
⇒ Caching effects at 1.98 GiB input size

Evaluation: Chunking

- **Chunking:** Split input data into chunks and asynchronous dispatch them

⇒ Already process some chunks while others are still being wired

- Chunked and whole strategies with regex `a[^b]{30}b`:



⇒ IndirectCommandBufferStrategy benefits from chunking

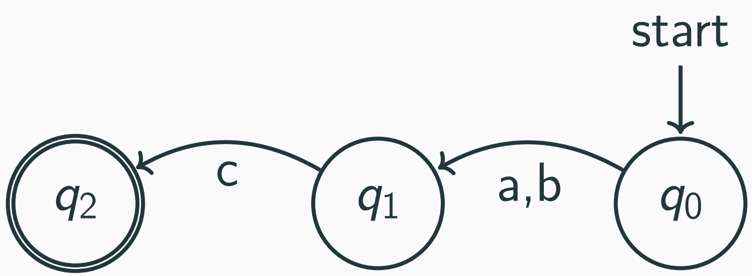
⇒ BasicStrategy shows no improvement.

Summary of Contribution & Future Work

- **JIT-Compilation:**
  - Critical optimization, nearly halved execution time.
- **Unified Memory:**
  - Initial wiring overhead exists.
  - Asynchronous dispatching mitigates wiring latency.
- **Performance:**
  - Throughput: GPU significantly outperforms multi-threaded CPU (up to 4x).
  - Scalability: GPU supports higher throughput for larger inputs.
  - Chunking: Effective only with low-overhead dispatch.
- **Future Work:**
  - Explore memory layout optimizations.
  - Test on different Apple Silicon chips.
  - Integrate with DuckDB for real-world applications.

Shift-And Algorithm: Example

Finite automata for the regex `[ab]c`



Bitmask representation:

- Mask Final: 10_2
- Mask Initial: 01_2
- Mask Char[a]: 01_2
- Mask Char[b]: 01_2
- Mask Char[c]: 10_2

Shift-And-Dist Algorithm

Algorithm 4 Shift-And-Dist algorithm

Require: $masks\_char[]$, $mask\_initial$, $mask\_final$, $text$

Require: $masks\_dist[]$ ▷ bit vectors for distance transitions

```
state ← 0
n_matches ← 0
for c in text do
    next ← mask_initial
    for d in 0 ... len(masks_dist) − 1 do        ▷ add all possible next states
        next ← next OR ((state AND masks_dist[d]) ≪ d)
    state ← next AND masks_char[c]
    if (state AND mask_final) ≠ 0 then
        n_matches ← n_matches + 1
return n_matches
```

Evaluation: Performance – Long Pattern

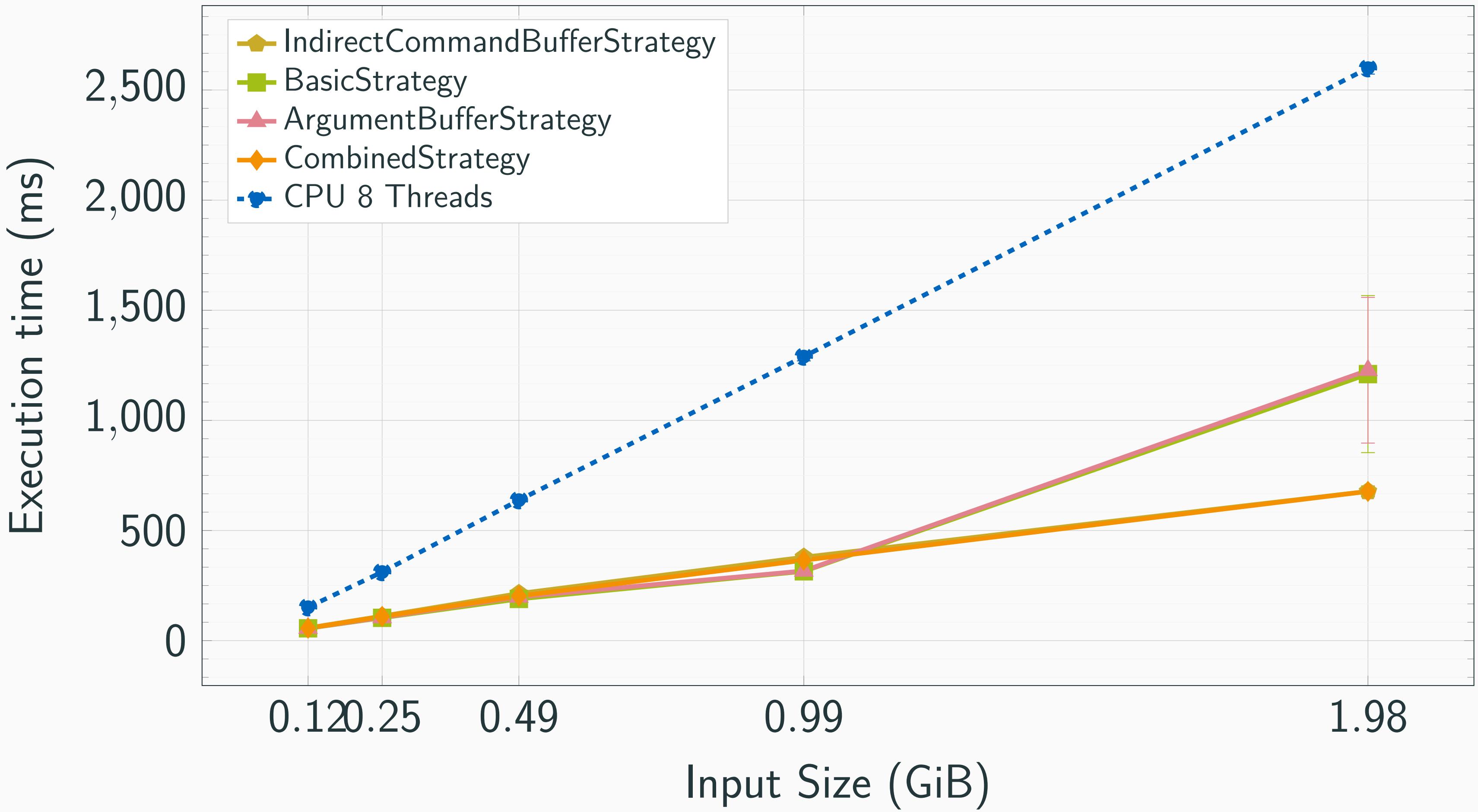


Figure 1: Execution times of the regex `a[^b]62b` on different large input sizes

Evaluation: 2GiB Threshold & Caching Impact

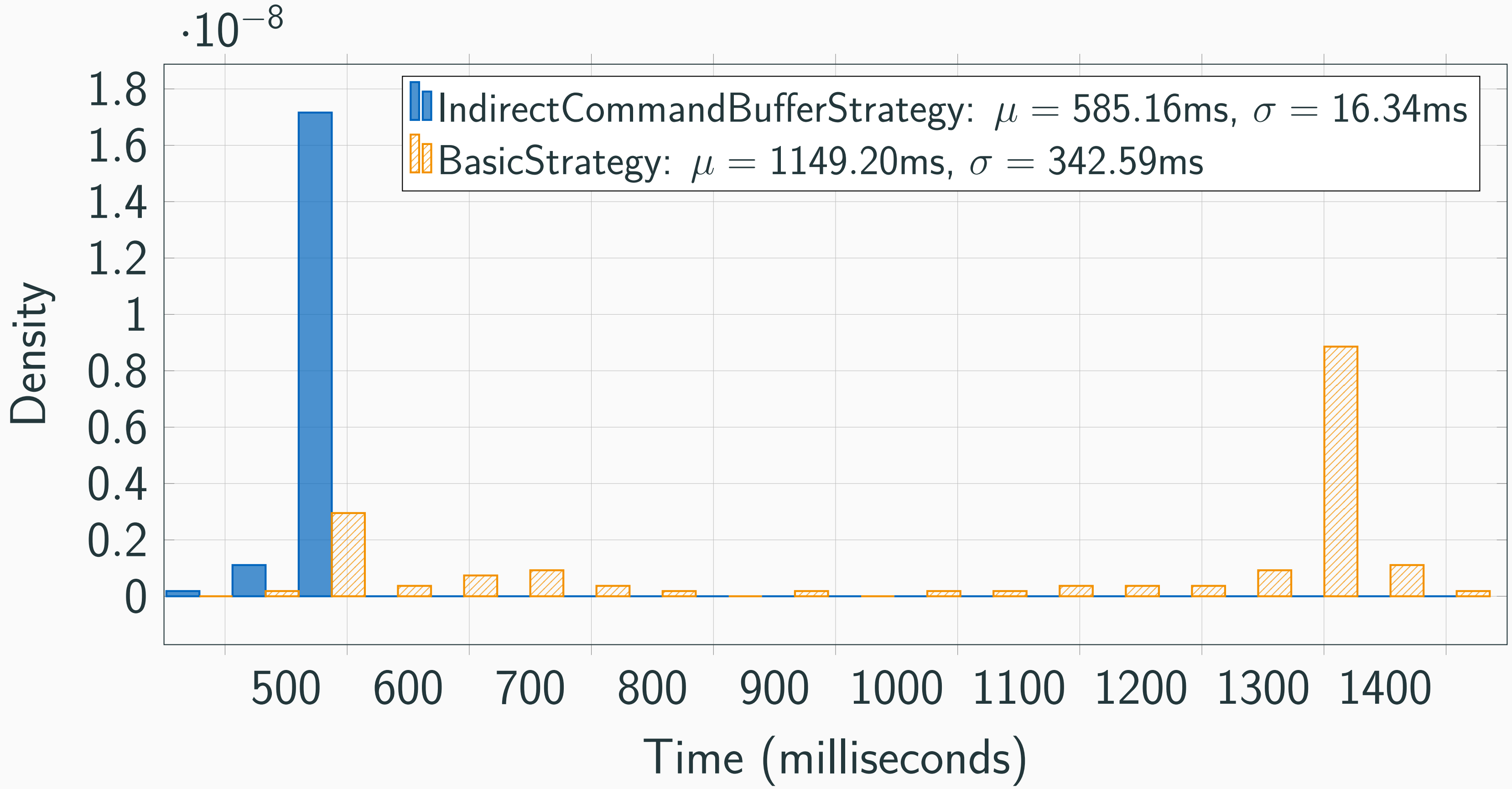


Figure 2: Time distribution comparison between IndirectCommandBufferStrategy and BasicStrategy when not chunked with 1.98GiB of input data

Evaluation: Performance – Large Jumping Distances

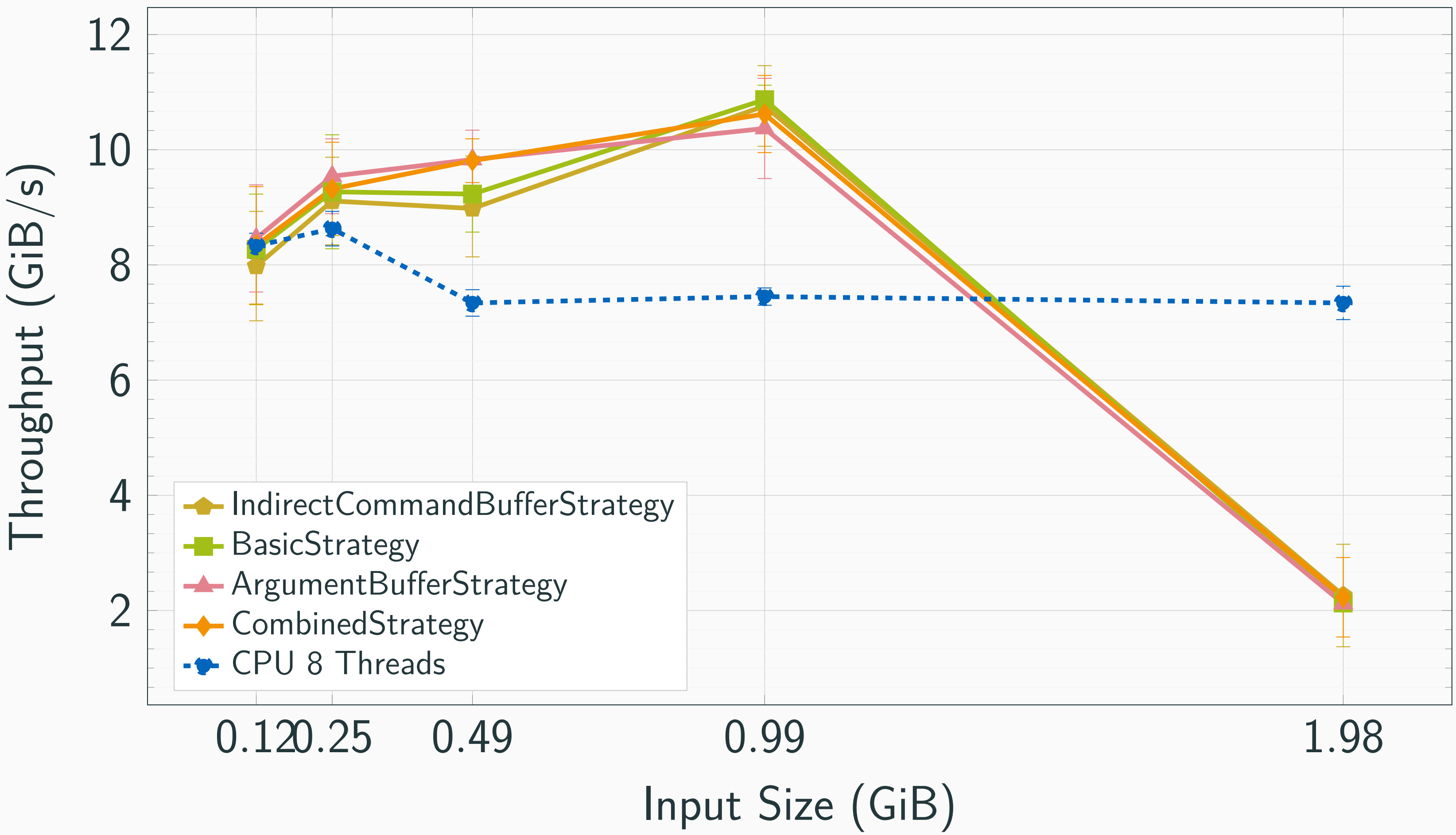


Figure 3: Throughput performance with the regex `a[^b]0,62b` on different input sizes