MICRO, October 20, 2025, Seoul, Korea

Interleaved Bitstream Execution for Multi-Pattern Regex Matching on GPUs

Tianao Ge[1], Xiaowen Chu[1], Hongyuan Liu[2]

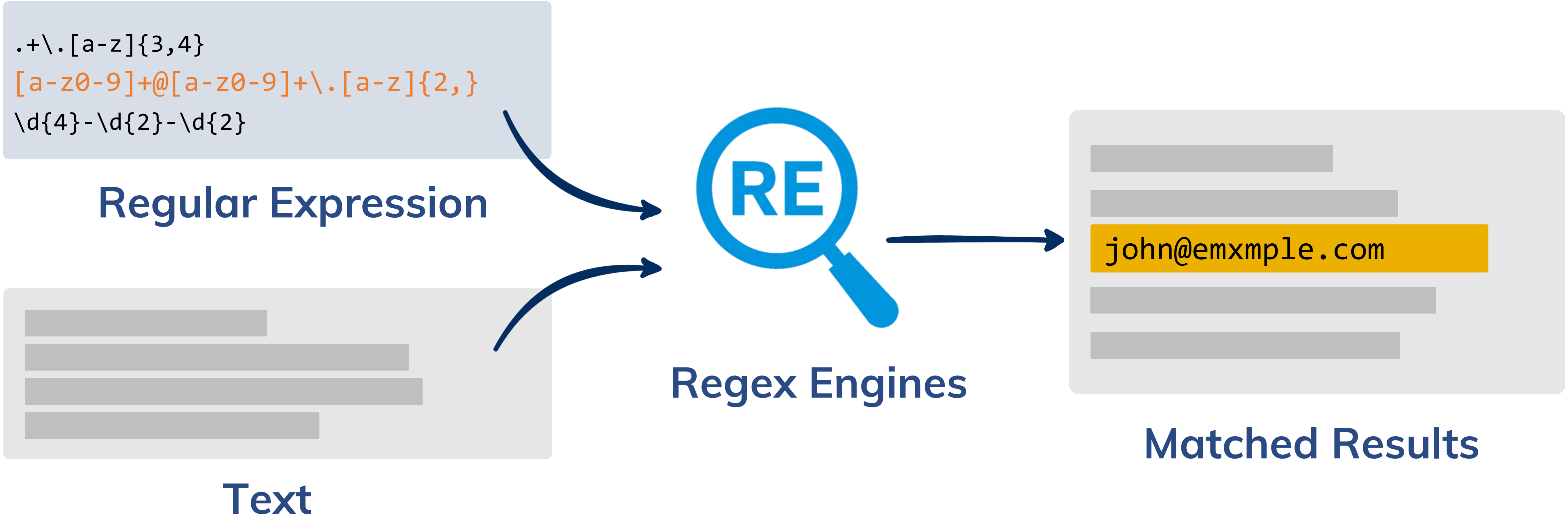
[1] Hong Kong University of Science and Technology (Guangzhou)

[2] Stevens Institute of Technology

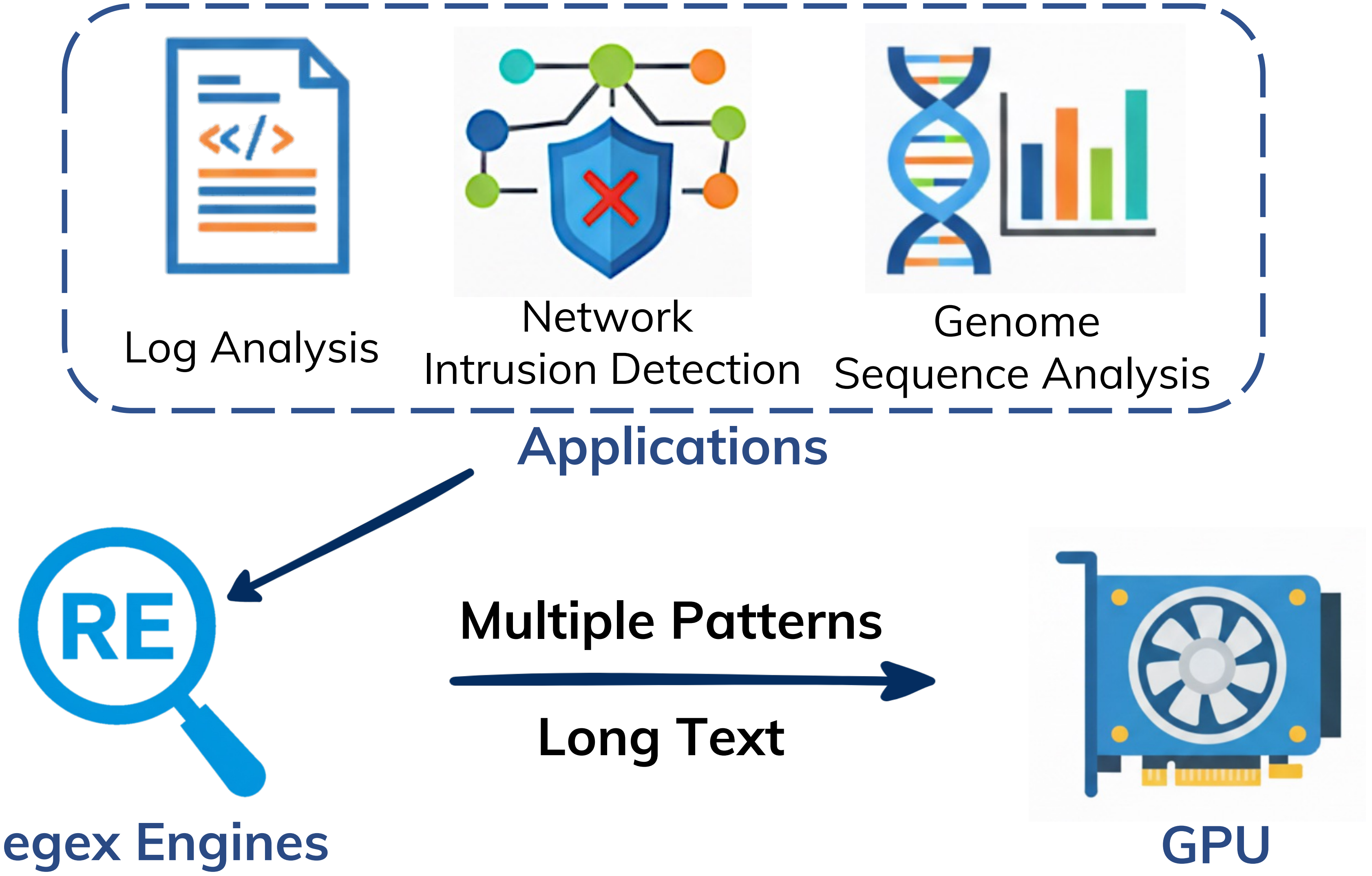
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

STEVENS INSTITUTE of TECHNOLOGY THE INNOVATION UNIVERSITY®

Regex Matching

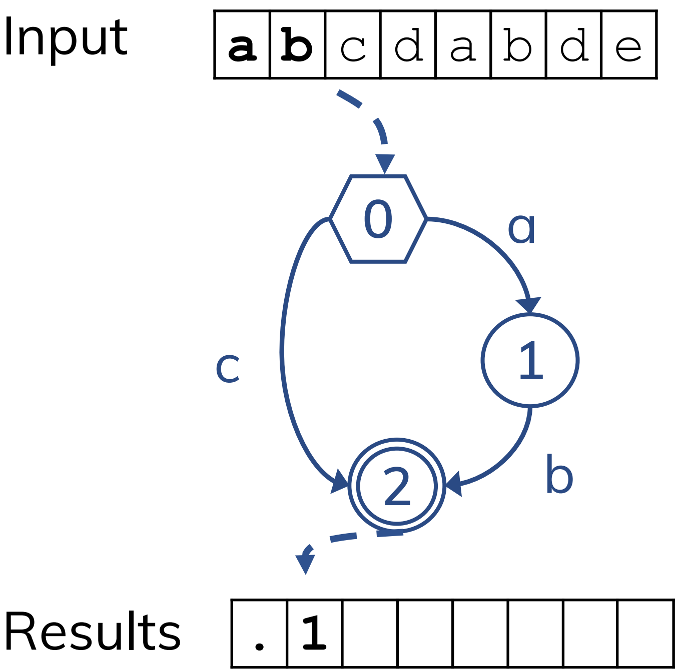


GPU-Accelerated Regex Engines

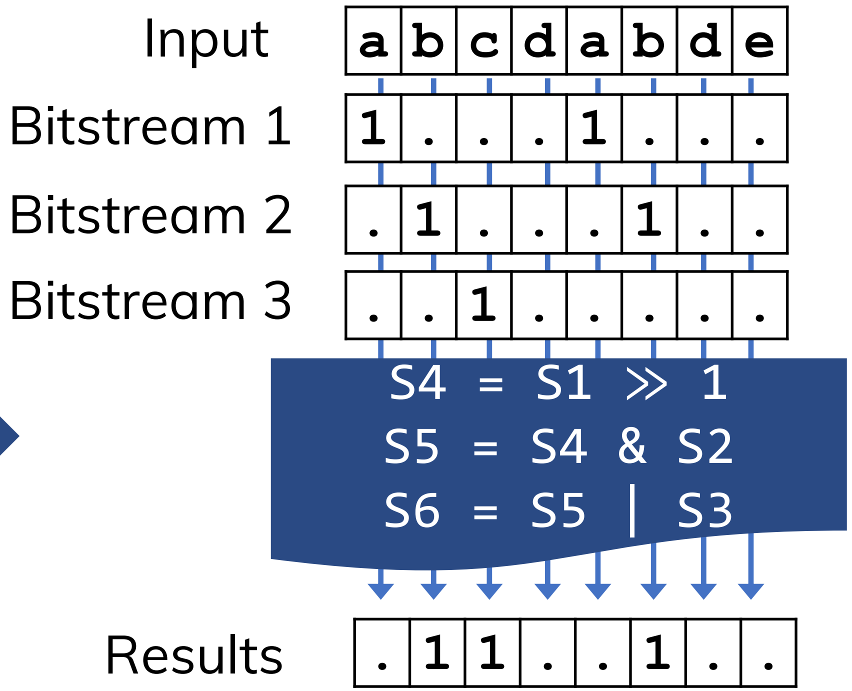


Execution Models for Regex Matching

Automata



Bitstream Program



Regex:

ab|c

! One Byte at a Time

! Irregular Access Patterns

✓ Regular Access

✓ Bit-Level Parallelism

? Data Dependencies

? Synchronization Overhead

? Bitstream Sparsity

Outline

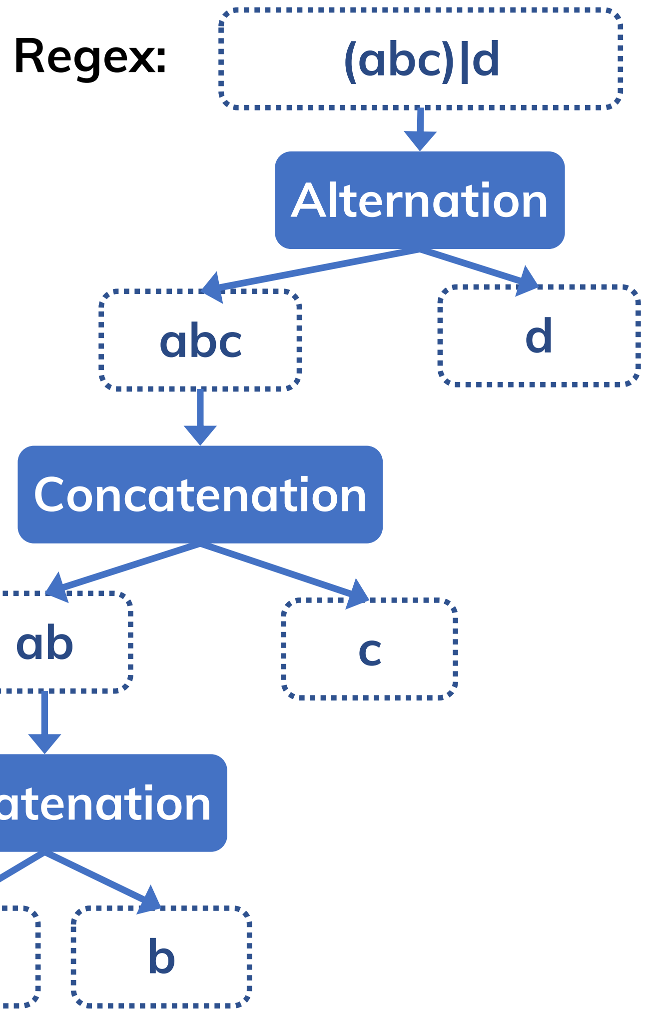
- Background:
  - Regex to Bitstream Program
  - Bitstream Execution on GPUs
- BitGen: Compiler for Bitstream Programs
  - Dependency-Aware Thread Data Mapping
  - Shift Rebalancing
  - Zero Block Skipping
- Performance Results

Regex to Bitstream Program: Step 1

Step 1: Parse regex

```
R ::= CC     // character class (e.g., a)
    | RR     // concatenation
    | R|R    // alternation
    | R*     // kleene star
    | R{n,m} // bounded repetition
    | R+     // one or more repetitions
    | R?     // zero or one repetition
```

Simplified Grammar of Regex



Abstract Syntax Tree

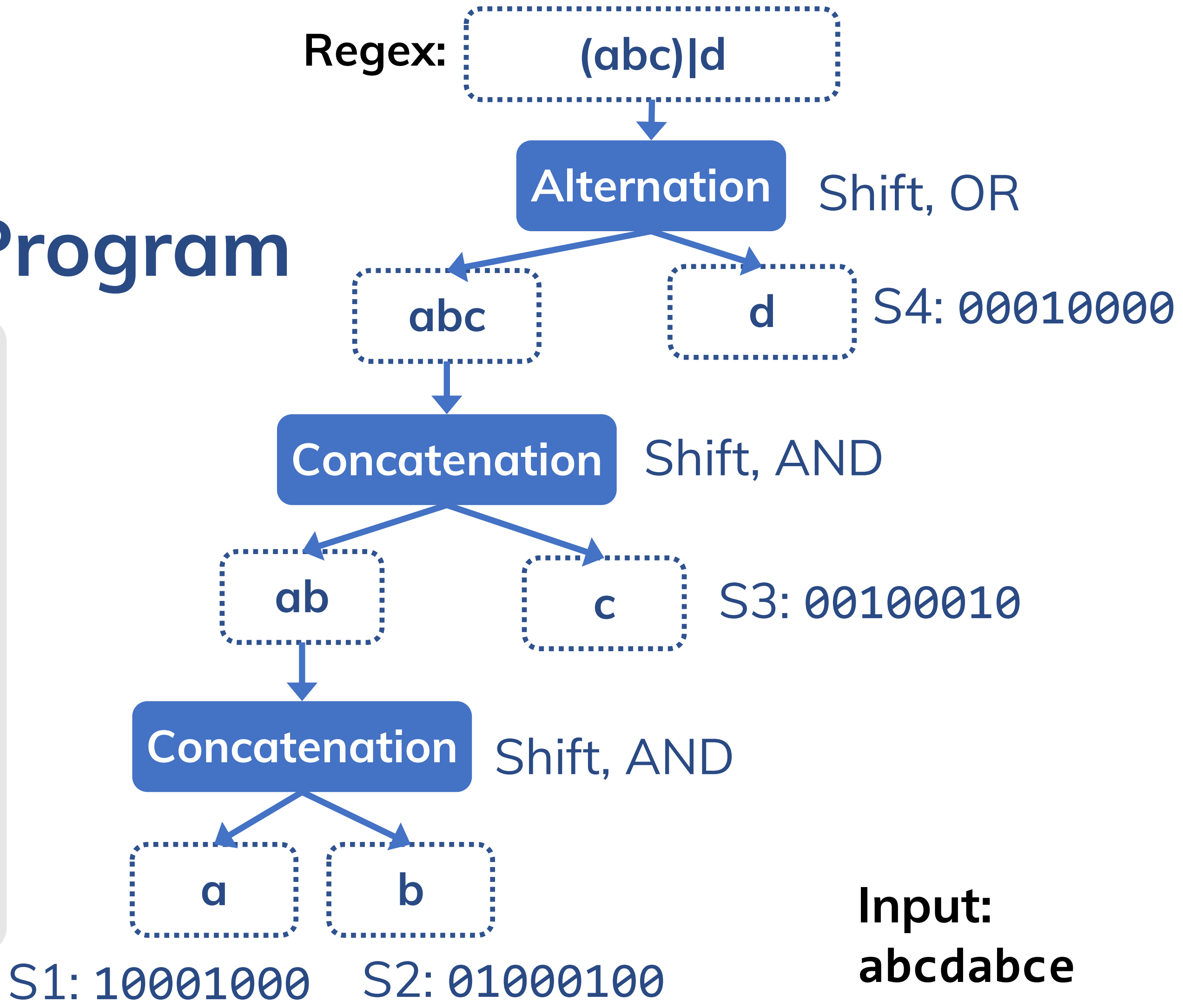
Regex to Bitstream Program: Step 2

Step 1: Parse regex

Step 2: Construct Bitstream Program

```
1.Character Class:
   a matching bitstream.
2.Concatenation:
   Shift and AND
3.Alternation:
   Shift and OR
4.Bounded Repetition:
   two for loops with Shift and AND
5.Kleene Star:
   a while loop with Shift and AND
```

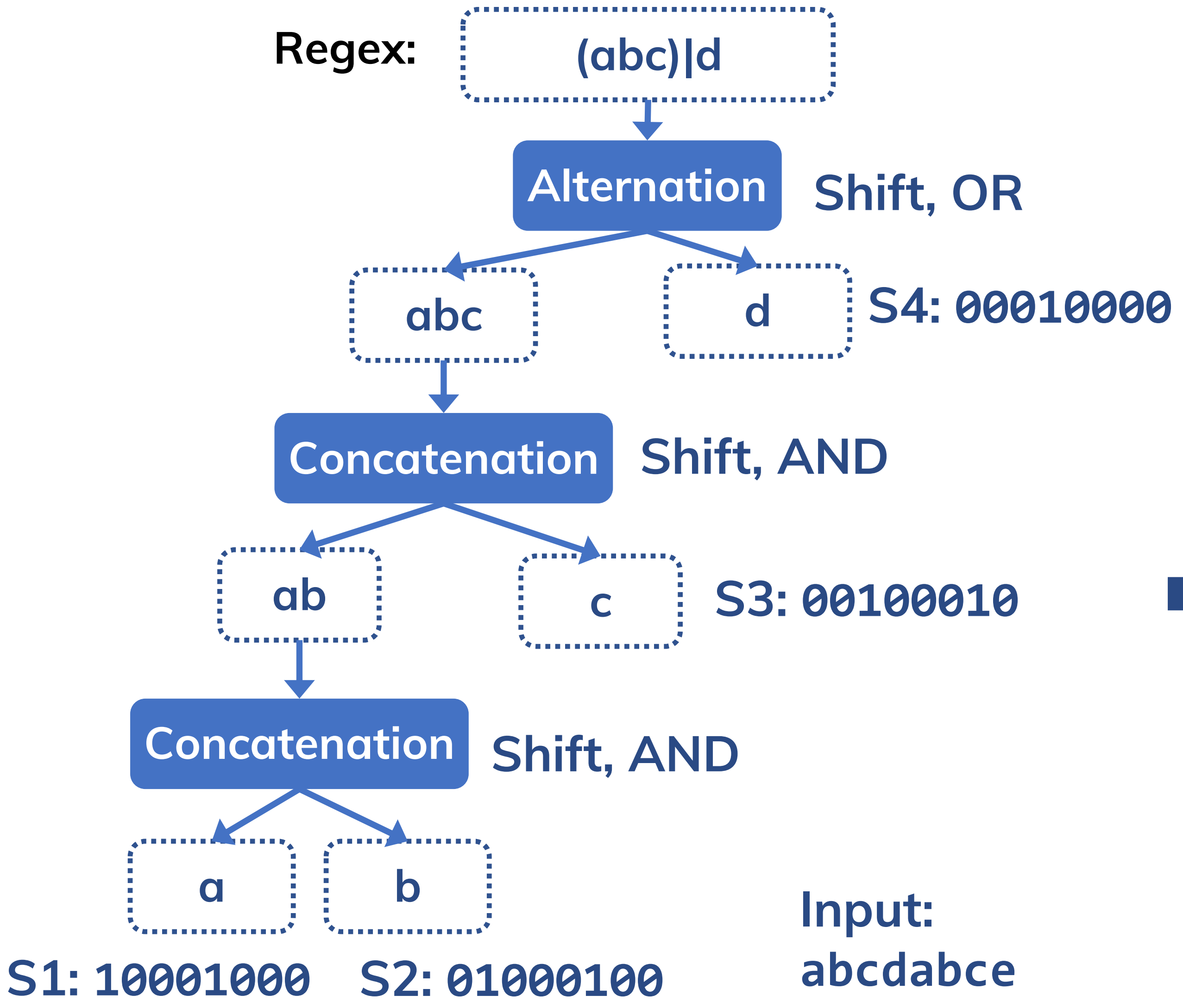
Regex to Bitstream Lowering Rules



Input:
abcdabce

Abstract Syntax Tree

Regex to Bitstream Program: Example



Abstract Syntax Tree

```
# input: text. output: S9
S1, S2, S3, S4 =
      match(text_trans, CCs)
S5 = S1 >> 1
S6 = S5 & S2          # ab
S7 = S6 >> 1
S8 = S7 & S3          # abc
S9 = S8 | S4          # abc|d
```

Bitstream Program

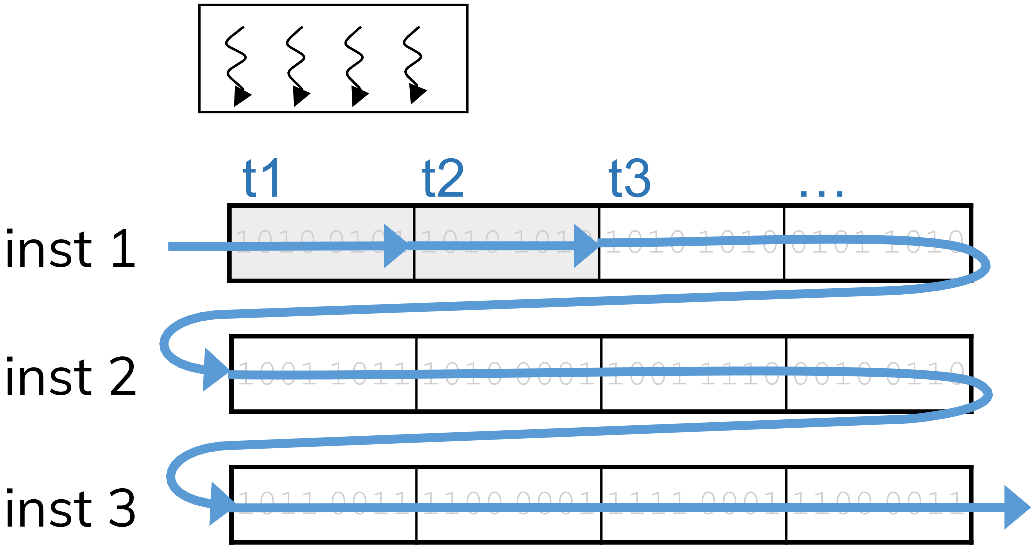
Another Example with Kleene Star

Regex:
a(bc)*d

```
# input: text
# output: S12
S1, S2, S3, S4 = match(text_trans, CCs) # a, b, c, d
S10 = S1
while (S1):
    S5 = S1 >> 1
    S6 = S2 & S5
    S7 = S6 >> 1
    S8 = S3 & S7
    S9 = ~S10
    S1 = S8 & S9
    S10 = S10 | S8                          # a(bc)*
S11 = S10 >> 1
S12 = S4 & S11                              # a(bc)*d
```

Bitstream Program

Bitstream Execution on GPUs

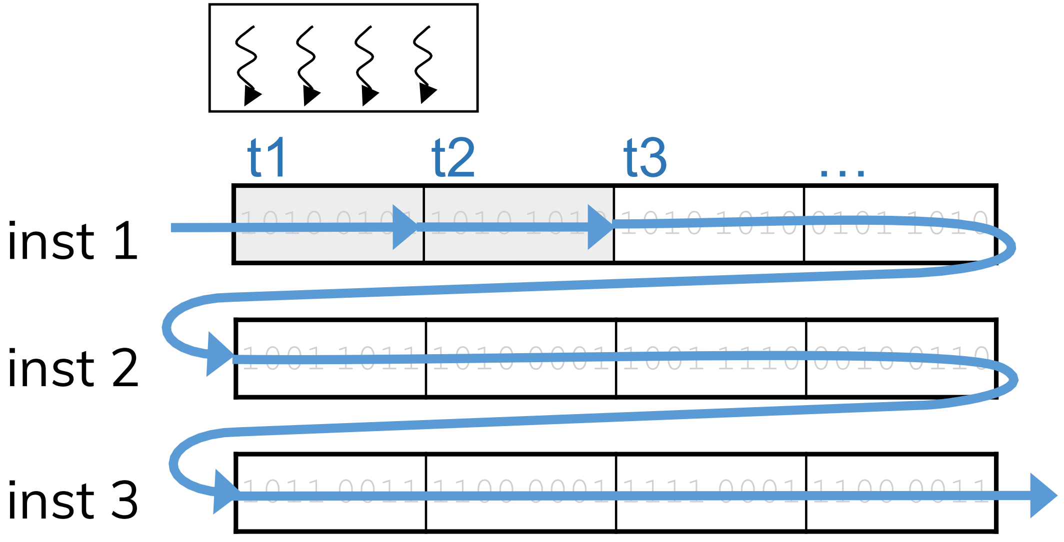



```
for inst in insts:
    for block in bitstreams:
        # execute inst on block
    # sync
```

✗ **Poor data reuse**
✗ **High memory usage**

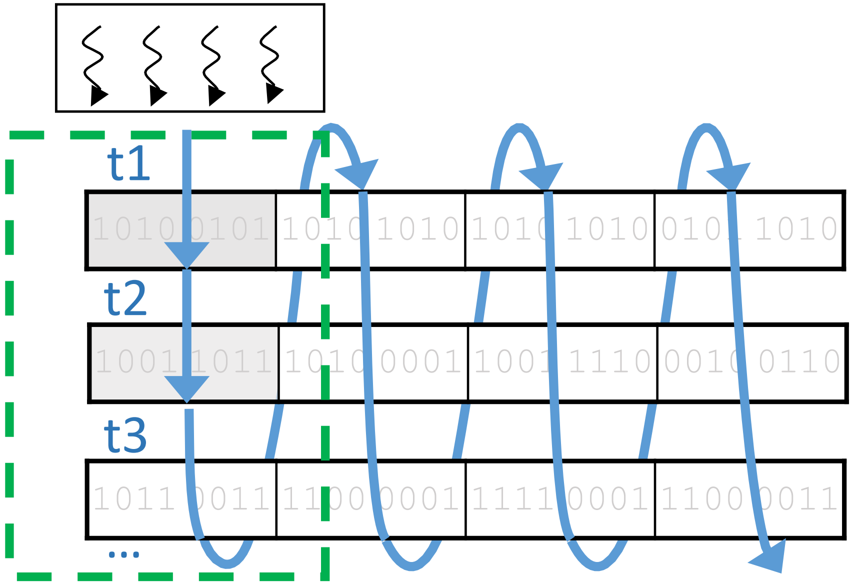
Sequential Execution

Key Insight: Interleaved Execution




```
for inst in insts:
    for block in bitstreams:
        # execute inst on block
    # sync
```

Sequential Execution

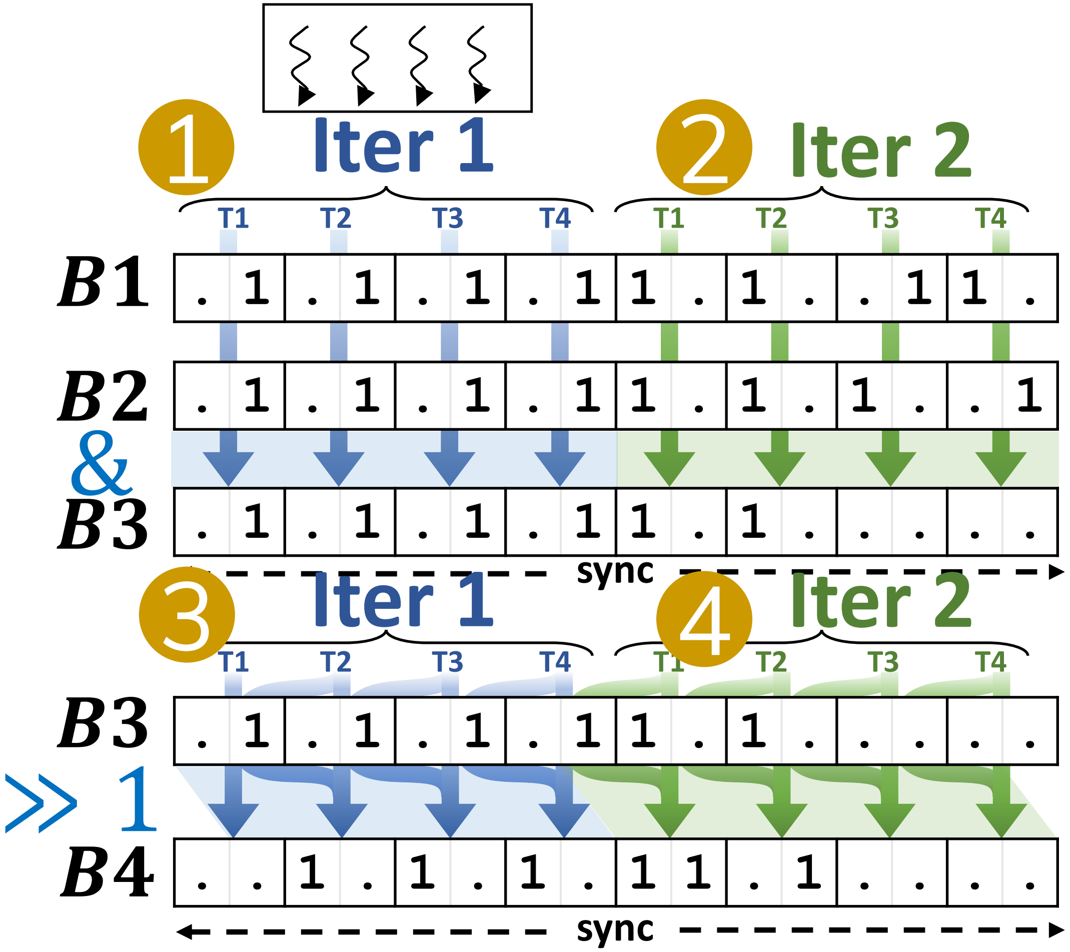



```
for block in bitstreams:
    for inst in insts:
        # execute inst on block
        # sync
```

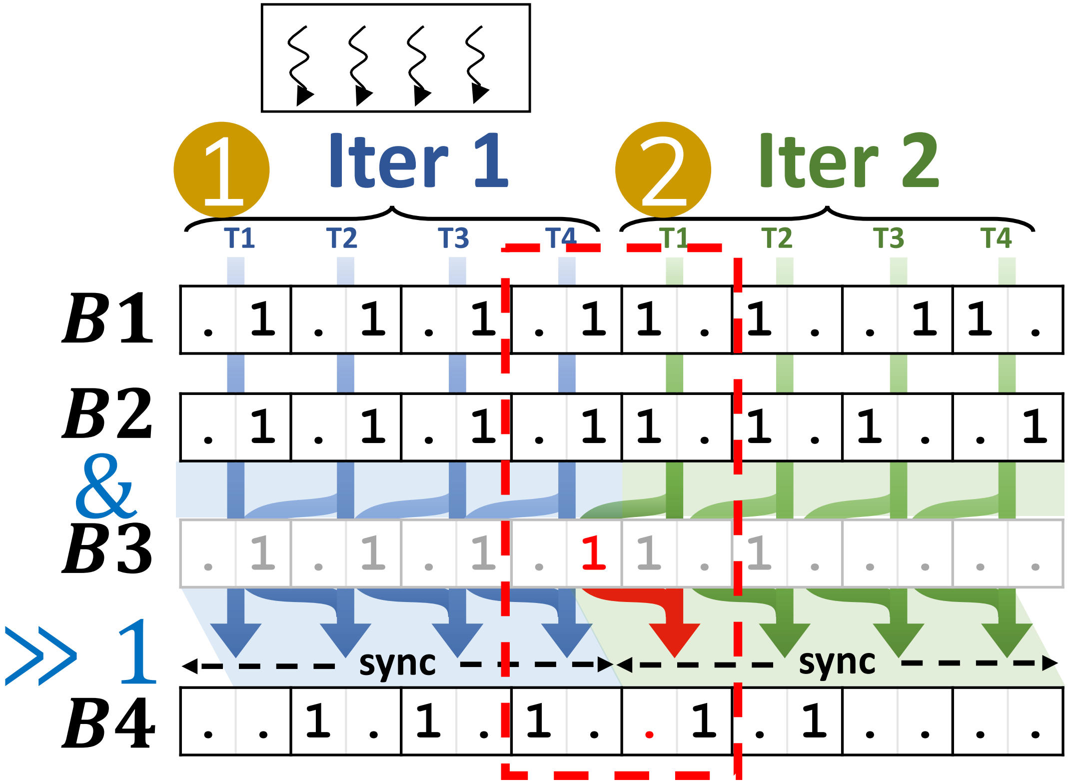
Interleaved Execution

✓ Data Reuse via Register

Challenge: Bit Dependencies in Shift



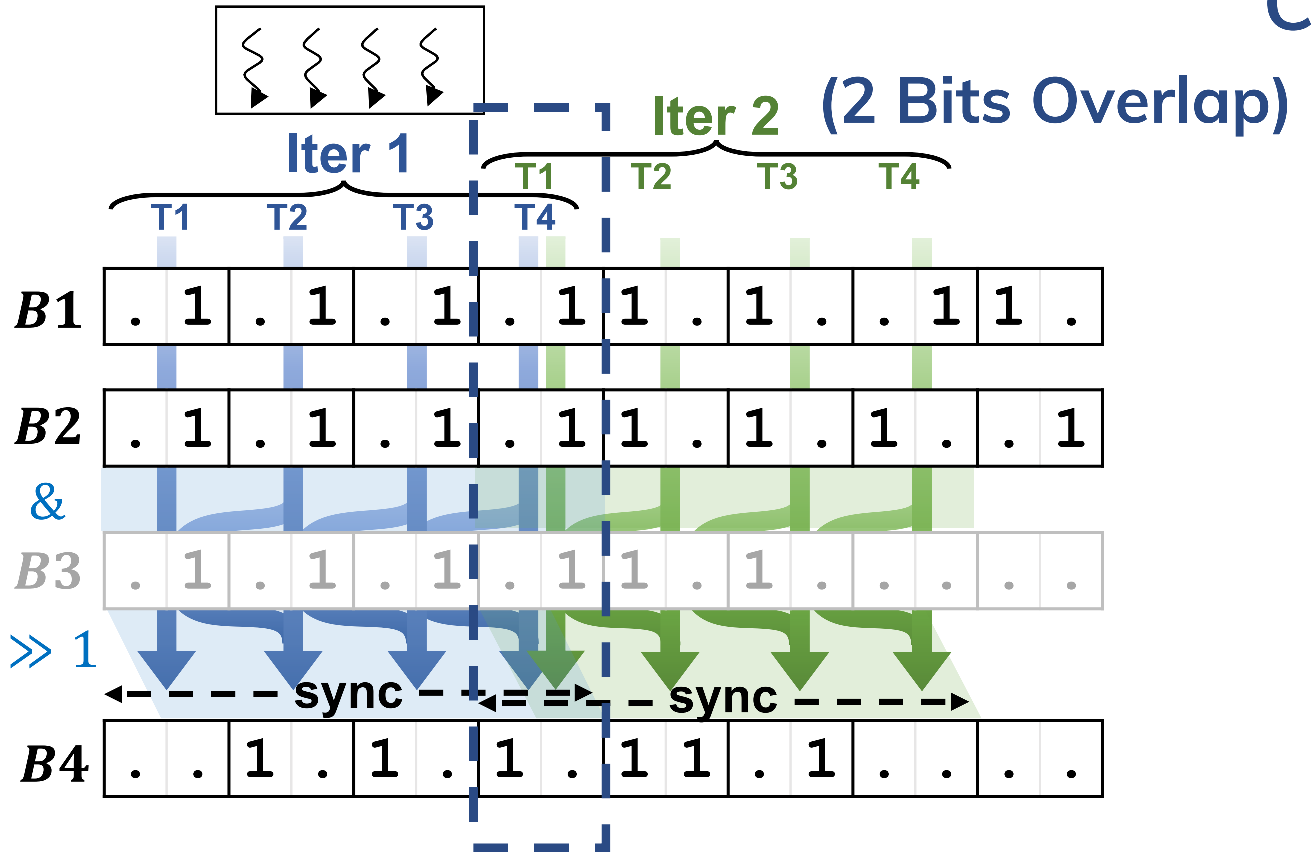
Sequential Execution



Error: Bits no available

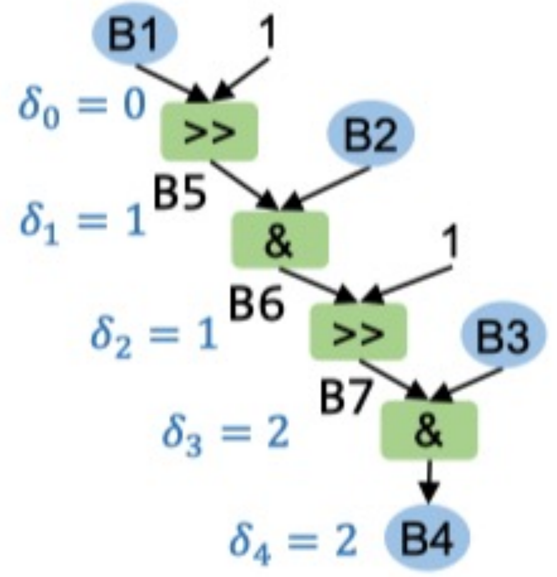
Interleaved Execution

Dependency-Aware Thread-Data Mapping



Retrieving Missing Bits via Recomputing

Calculating Overlap Distance Δ



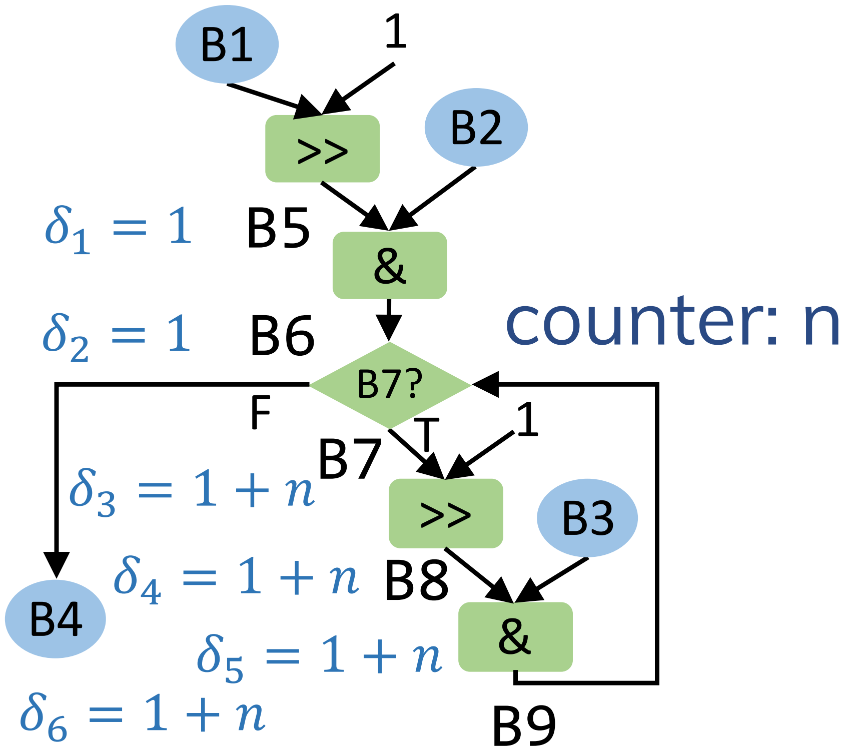
$$\delta_{i+1} = \delta_i + k_i$$

$$\Delta = \max_{P \in \mathrm{Paths}} \left(\max_i \delta_i - \min_i \delta_i \right)$$

δ: cumulative bit offset
k: signed shift distance

Dependency-Aware Thread-Data Mapping

How about while loop?



Computing Overlap Distance Δ

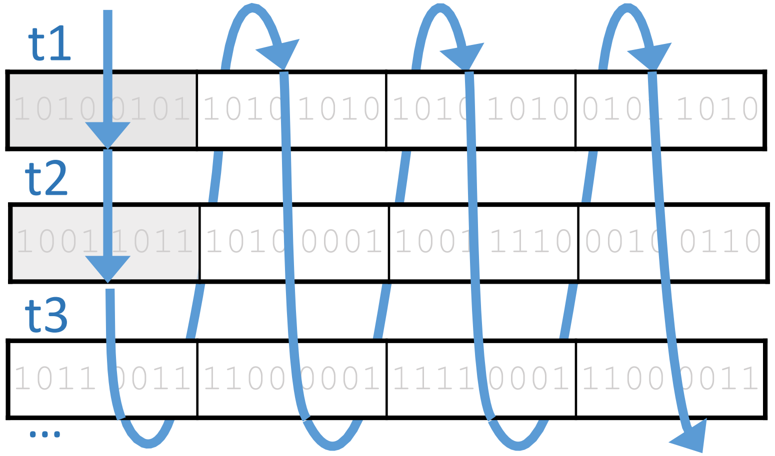
$$\delta_{i+1} = \delta_i + \mu_s(n_1, \ldots, n_L) \cdot k_s$$

$$\Delta(n_1, \ldots, n_L) = \max_{P \in \text{Paths}} \left(\max_i \delta_i - \min_i \delta_i \right)$$

δ: cumulative bit offset
k: signed shift distance
μ: multiplicity function
n: loop iteration counters

Synchronization Overhead

- Shift Operations
- Conditional Operations: if, while




```
for block in bitstreams:
    for inst in insts:
        # execute inst on block
        # sync
```

Interleaved Execution

Two consecutive SHIFT with syncs

4 Syncs

```
sync                                        Thread
smem1[tid] = B2
sync
B5 =[smem1[tid-1], smem1[tid]] ≪ 1
```

```
sync                                        Thread
smem1[tid] = B3
sync
B7 =[smem1[tid-1], smem1[tid]] ≪ 2
```

2 Syncs

```
sync                                        Thread
smem1[tid] = B2
smem2[tid] = B3
sync
B5 =[smem1[tid-1], smem1[tid]] ≪ 1
B7 =[smem2[tid-1], smem2[tid]] ≪ 2
```

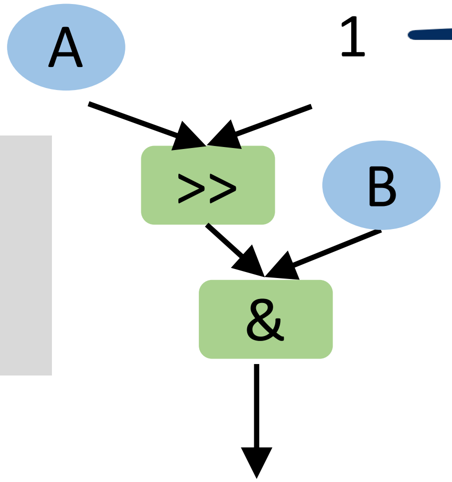
Solution: Merge Sync Points

But how to schedule SHIFT together?

Breaking Long Dependency Chains

`(A ≫ 1) & B`

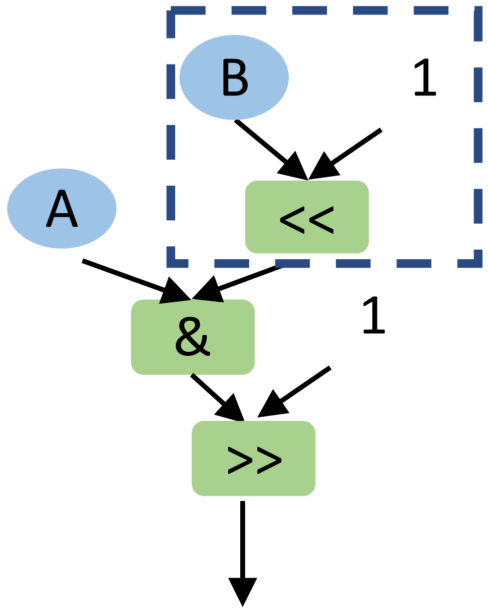
```
C = A ≫ 1
D = C & B
```



Operand Rewriting

`(A & (B ≪ 1)) ≫ 1`

```
C = B ≪ 1
D = C & A
E = D ≫ 1
```

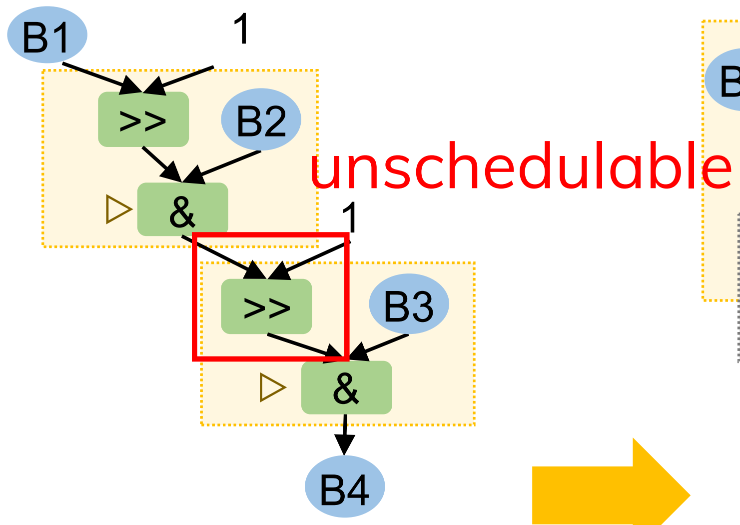


If A is not ready,
≫, & can't start

If A is not ready,
≪ can start

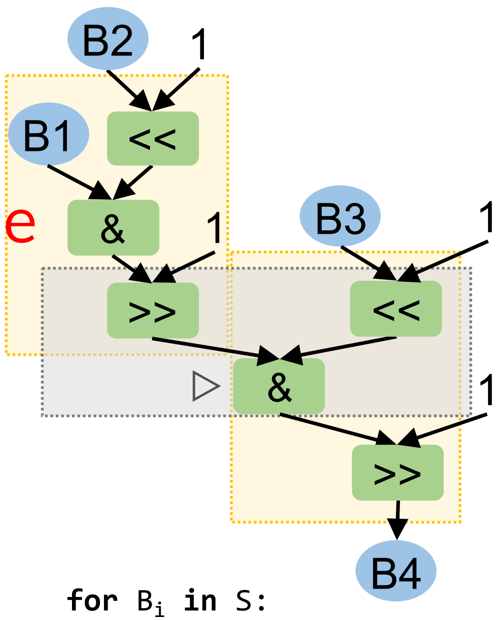
Operations on B can be scheduled before A

Shift Rebalancing



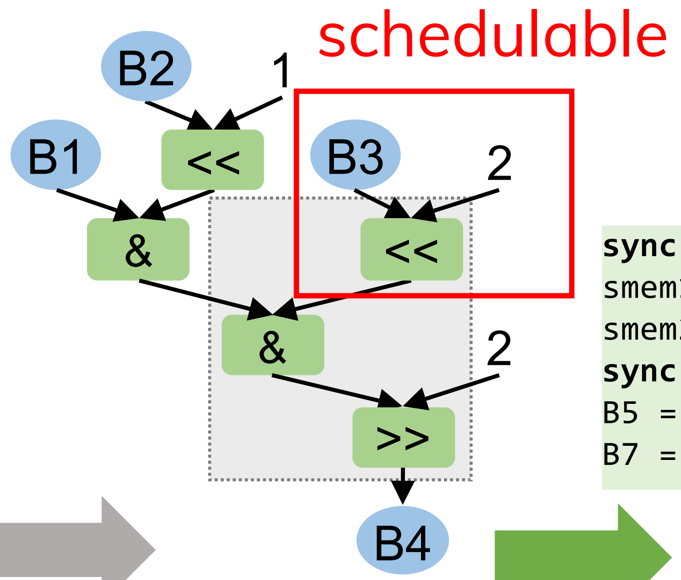

```
for Bi in S:
1   B4 = B1 ≫ 1
2   B5 = B4 & B2 ▷Rebalance
3   B6 = B5 ≫ 1
4   B7 = B6 & B3 ▷Rebalance
```

Original Code:
Shift rebalance at lines 2 & 4




```
for Bi in S:
1   B5 = B2 ≪ 1
2   B6 = B1 & B5
3   B7 = B6 ≫ 1
4   B8 = B3 ≪ 1
5   B9 = B7 & B8 ▷Rebalance
6   B4 = B9 ≫ 1
```

Iteration 1:
Shift rebalance at line 5




```
for Bi in S:
1   B5 = B2 ≪ 1
2   B6 = B1 & B5
3   B7 = B3 ≪ 2
4   B8 = B6 & B7
5   B4 = B8 ≫ 2
```

Iteration 2:
Converged

```
sync                                              Thread
smem1[tid] = B2
smem2[tid] = B3
sync
B5 =[smem1[tid-1], smem1[tid]] ≪ 1
B7 =[smem2[tid-1], smem2[tid]] ≪ 2
```

```
for Bi in S:
1 B5, B7 = B2 ≪ 1, B3 ≪ 2
2 B7 = B6 = B1 & B5
3 B8 = B6 & B7
4 B4 = B8 ≫ 2
```

Merged

Skipping Computation for Zero Bits

Short-circuiting operation:

AND operation

A & B

```
if A == 0:
    C = 0
else:
    C = A & B
```

SHIFT operation

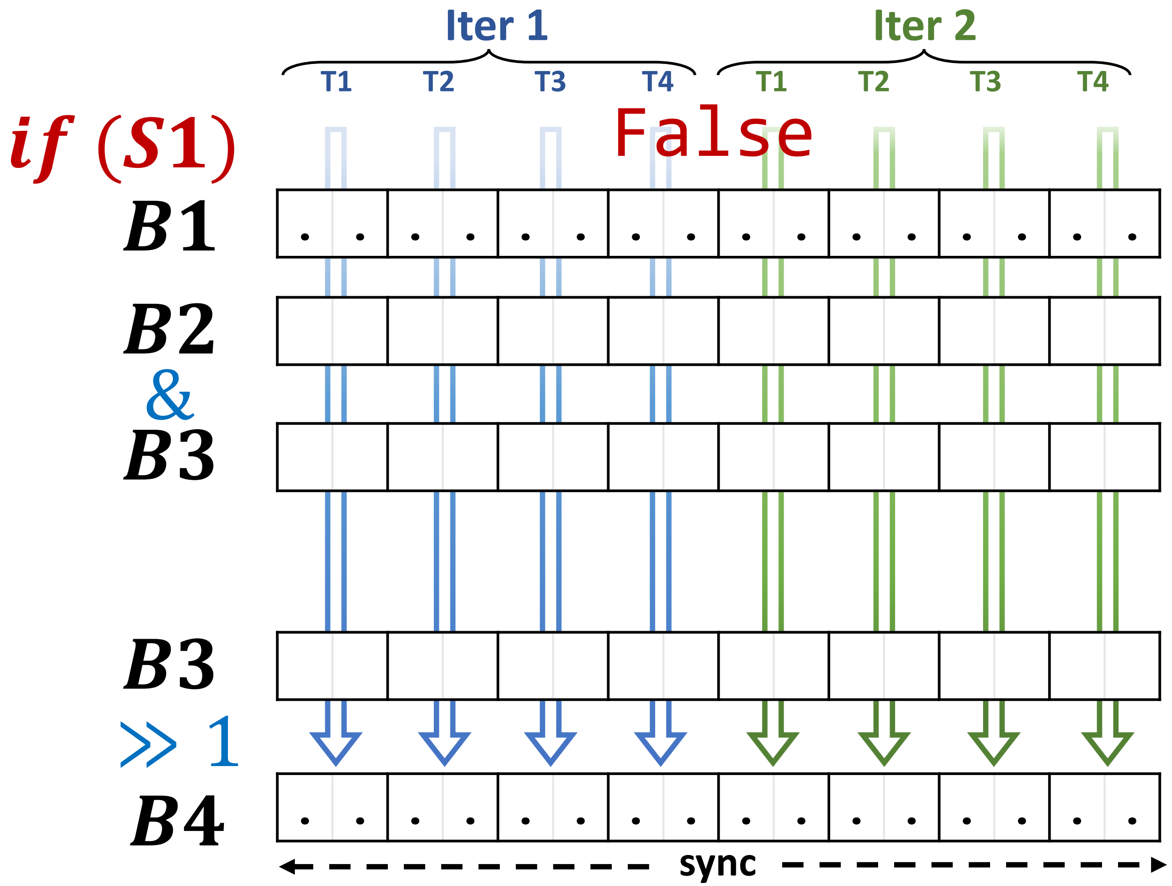
A ≫ 1

```
if A == 0:
    B = 0
else:
    B = A ≫ 1
```

Skipping Computation for Zero Bitstream

Bitstream-Level Zero Skipping

```
if (S1)
  S3 = S1 & S2
  S4 = S3 >> 1
else:
  S4 = 0
```

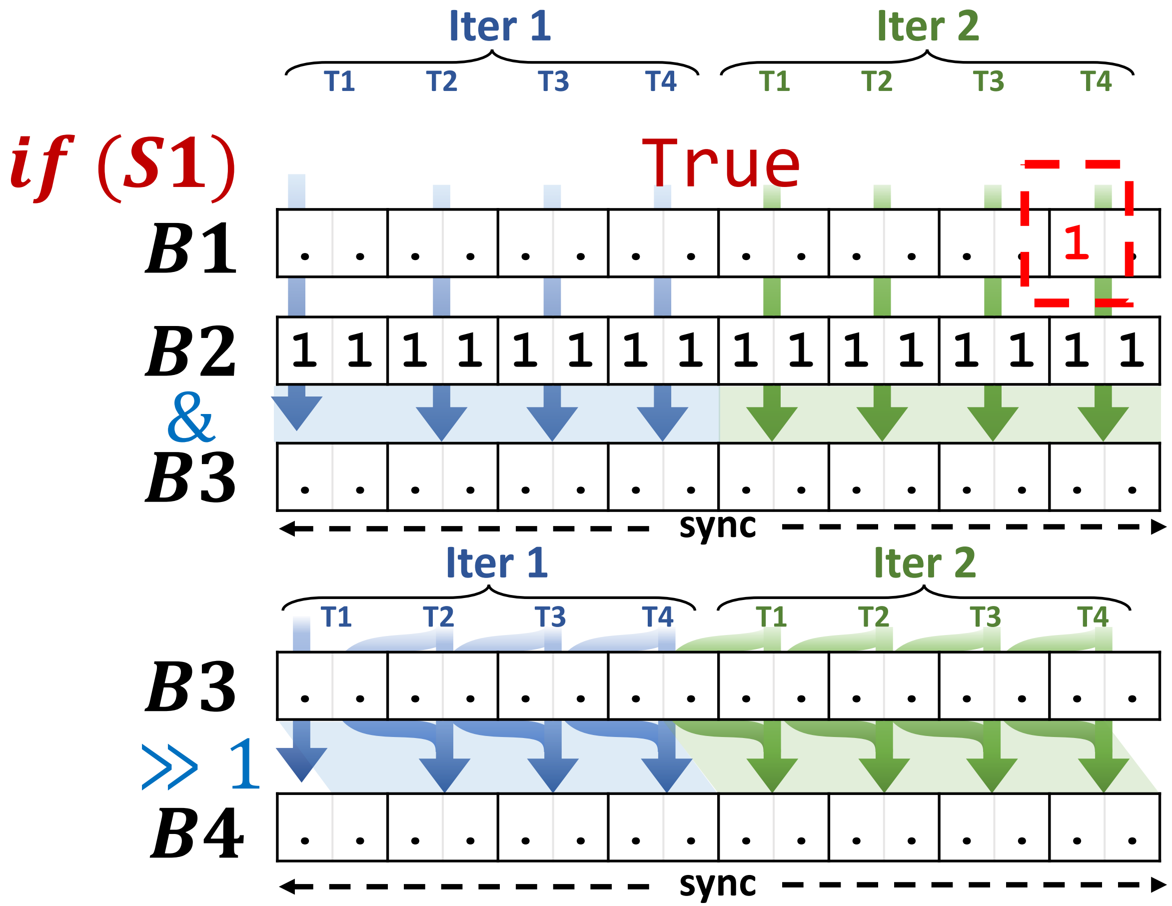


Sequential Execution

Skipping Computation for Zero Bitstream

Bitstream-Level Zero Skipping

```
if (S1)
  S3 = S1 & S2
  S4 = S3 >> 1
else:
  S4 = 0
```

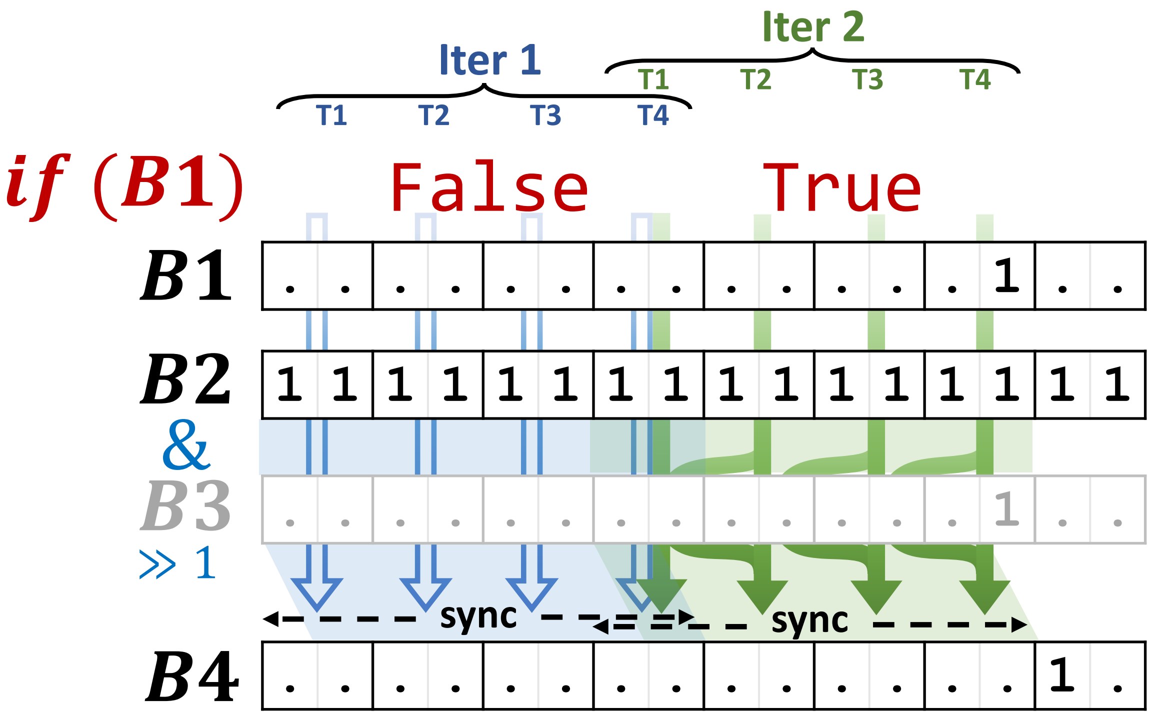


Sequential Execution

Sparsity in Interleaved Execution

Block-Level Zero Skipping

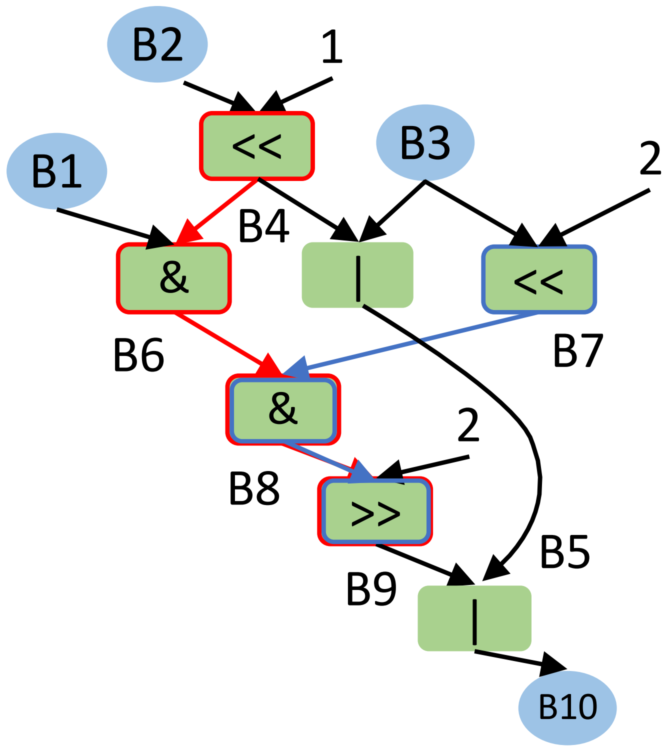
```
if (S1)
  S3 = S1 & S2
  S4 = S3 >> 1
else:
  S4 = 0
```



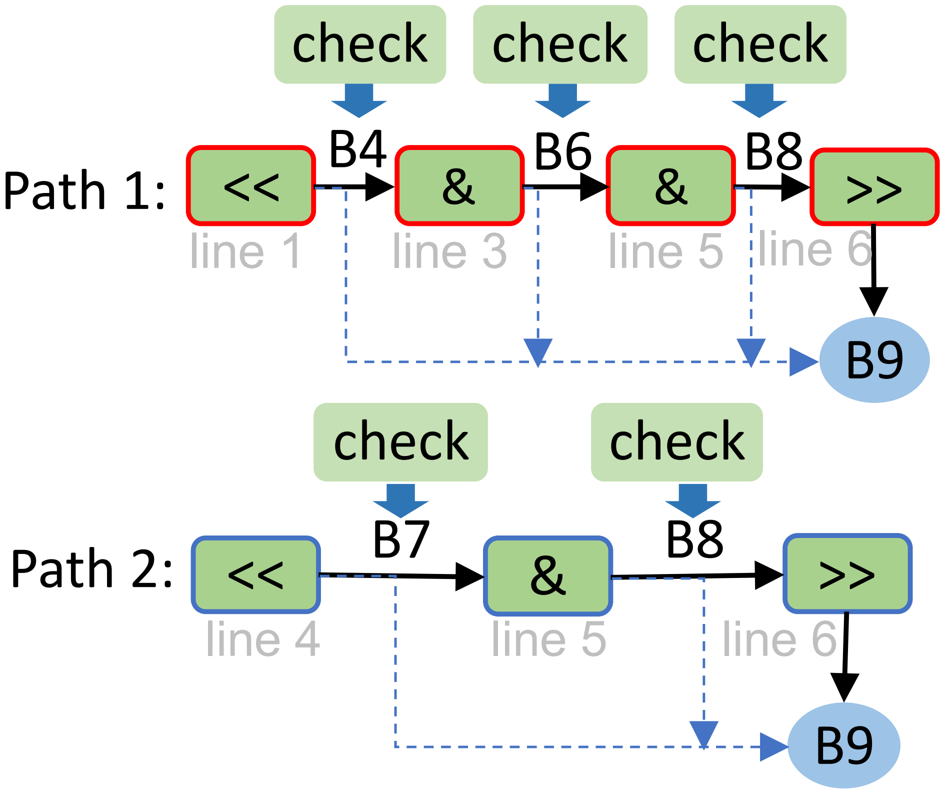
Interleaved Execution

Zero Block Skipping

1. Traversing the DFG
2. Identifying Zero Paths

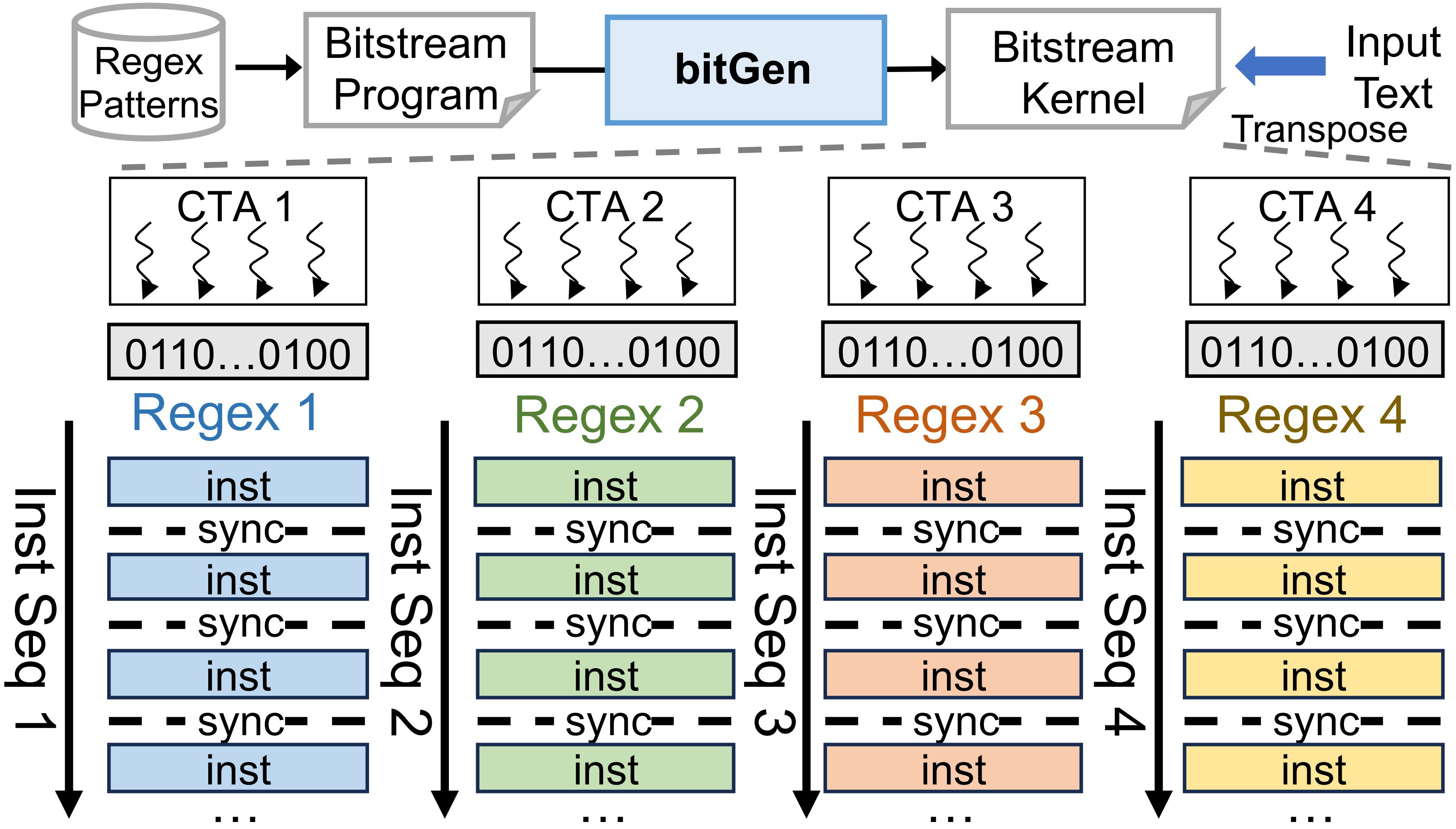


3. Inserting Branch



Implementation

BitGen: Compiler for Bitstream Programs



Evaluation

Baselines

- **ngAP** [1]: GPU, automata-based
- **icgrep** [2] : CPU, bitstream-based
- **Hyperscan**[3]: CPU,automata-based, multithreaded

Benchmarks

- **AutomataZoo and ANMLZoo**:
  Brill, ClamAV, Dotstar, Protomata, Snort
- **Regex**:
  Bro217, ExactMatch, Ranges1, TCP

Configuration

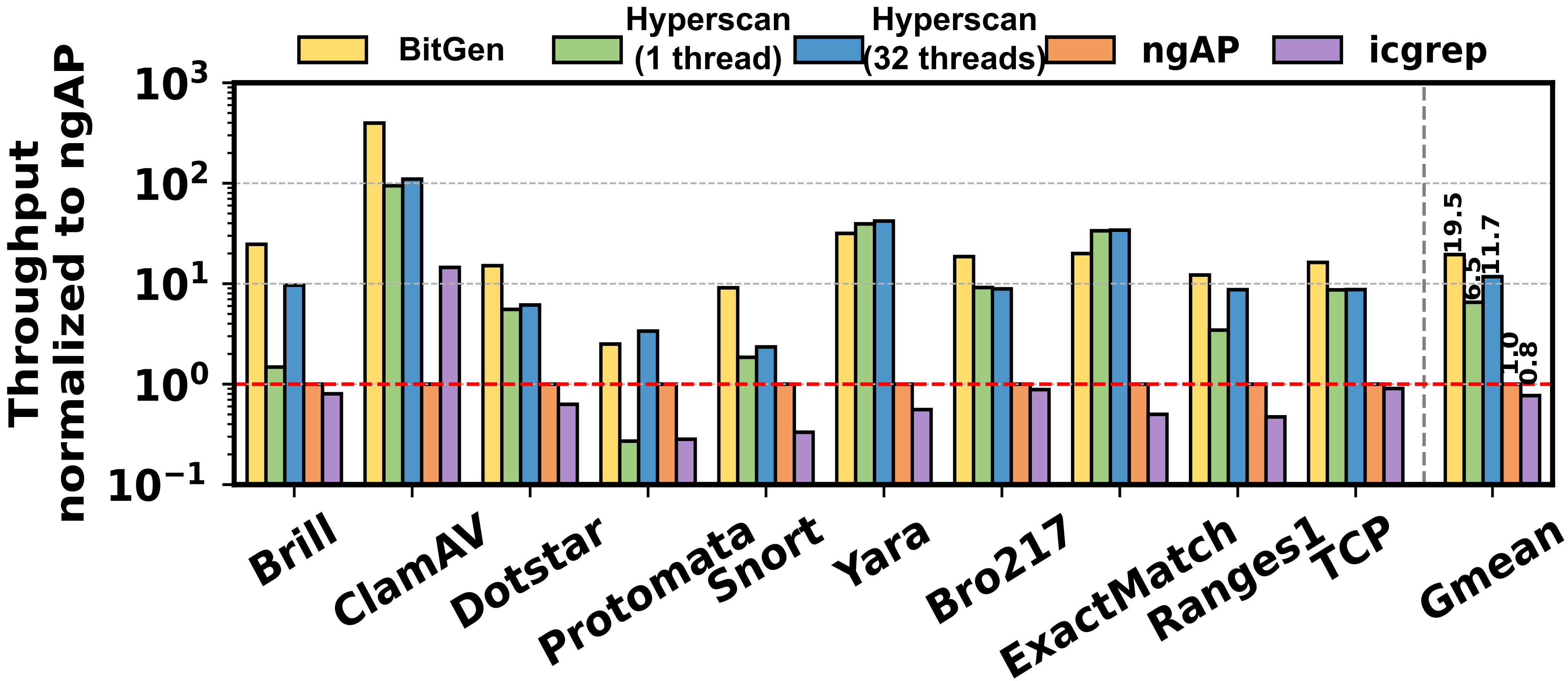
- **NVIDIA RTX 3090** (Ampere, 24 GB, 82 SMs)
- Intel Xeon Platinum 8562Y+ (32 cores)
- 128 GB memory
- CUDA 12.4 and GCC 13.2.

Metrics

- **Throughput** : as the number of input symbols processed per second (MB/s)

[1] Tianao Ge; et al. ngAP: Non-blocking Largescale Automata Processing on GPUs. ASPLOS 2024.
[2] Robert D. Cameron; et al. Bitwise data parallelism in regular expression matching. PACT 2014.
[3] Xiang Wang ; et al. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. NSDI 2019.

Overall Performance



BitGen improves performance by an average of 19.5× ngAP and 1.7× over Hyperscan running on 32 CPU threads

More Results

- **Performance Breakdown**
  - 12.0× → 17.6× → 24.9×
- **Prifiling Results**
  - DRAM Read/Writing: 263.1 MB → 0.4 MB
  - Barrier Stall : 49.6 % → 17.5 %
- **Sensitivity Studies**
  - the maximum number of syncs to be merged
  - the frequency of inserting conditional branches
- **Portability Studies**
  - 3090 v.s. H100 v.s. L40S

Conclusion

- **Optimized Bitstream Execution** for Regex Matching on GPUs.
- **Key Techniques**: GPU Code Generation with **Interleaved Execution**
  - Dependency-aware thread mapping to resolve dependencies
  - Shift rebalancing to reduce synchronization
  - Zero-block skipping to reduce redundant computation
- **Significant Performance Gains**
  - 19.5× and 1.7× geometric mean speedup over state-of-the-art GPU and CPU regex engines

BitGen Code Repository:

https://github.com/getianao/BitGen

GitHub Code

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

MICRO, October 20, 2025, Seoul, Korea

Thanks and Questions

Interleaved Bitstream Execution for Multi-Pattern Regex Matching on GPUs

Tianao Ge[1] <tge601@connect.hkust-gz.edu.cn>,

Xiaowen Chu[1], Hongyuan Liu[2]

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

STEVENS INSTITUTE of TECHNOLOGY THE INNOVATION UNIVERSITY®