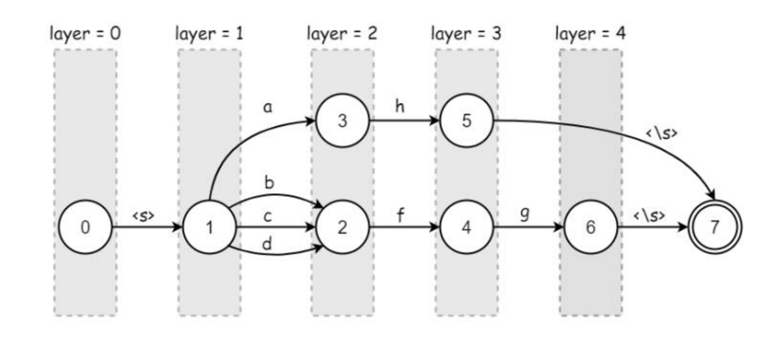
A GPU-based WFST Decoder with Exact Lattice Generation

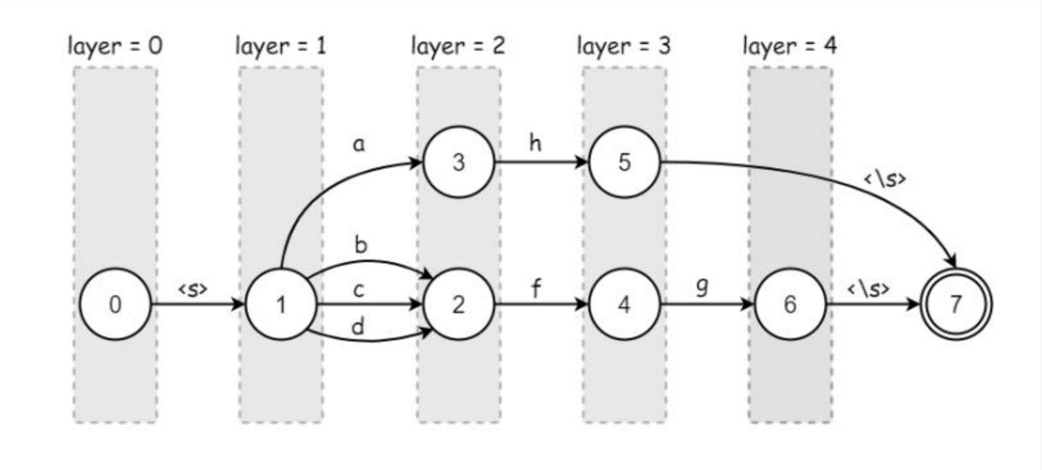
Zhehuai Chen, Justin Luitjens, **Hainan Xu, Yiming Wang**,
Daniel Povey, Sanjeev Khudanpur



The Center For Language and Speech Processing at the Johns Hopkins University
SJTU Speech Lab
上海交通大學智能語音實驗室
NVIDIA
KALDI

Outline

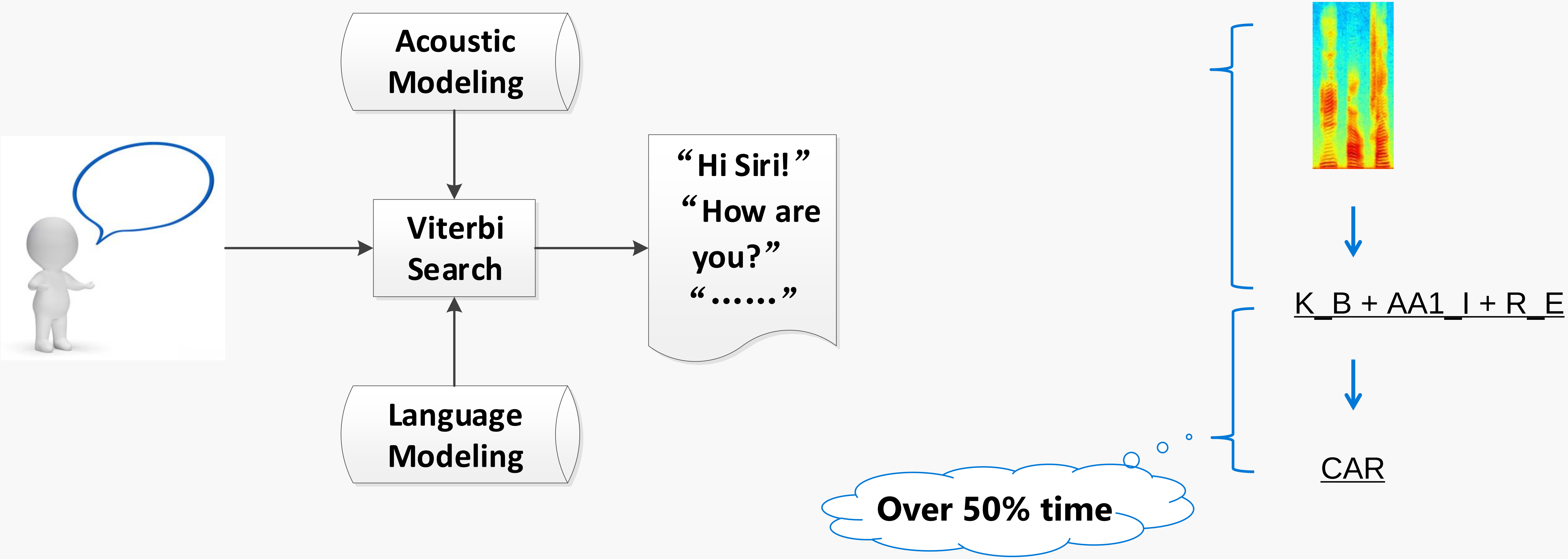
- Introduction
  - ASR and WFST Decoding
  - GPU-based Parallel Computing
- Parallel Viterbi Decoding[1]
  - Framework
  - Token Recombination
  - Load Balancing
  - Lattice Processing
- Experiments



1. https://github.com/chenzhehuai/kaldi/tree/gpu-decoder

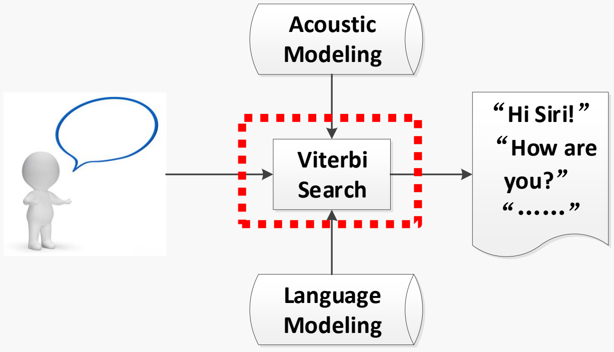
Introduction

- Automatic Speech Recognition (ASR)

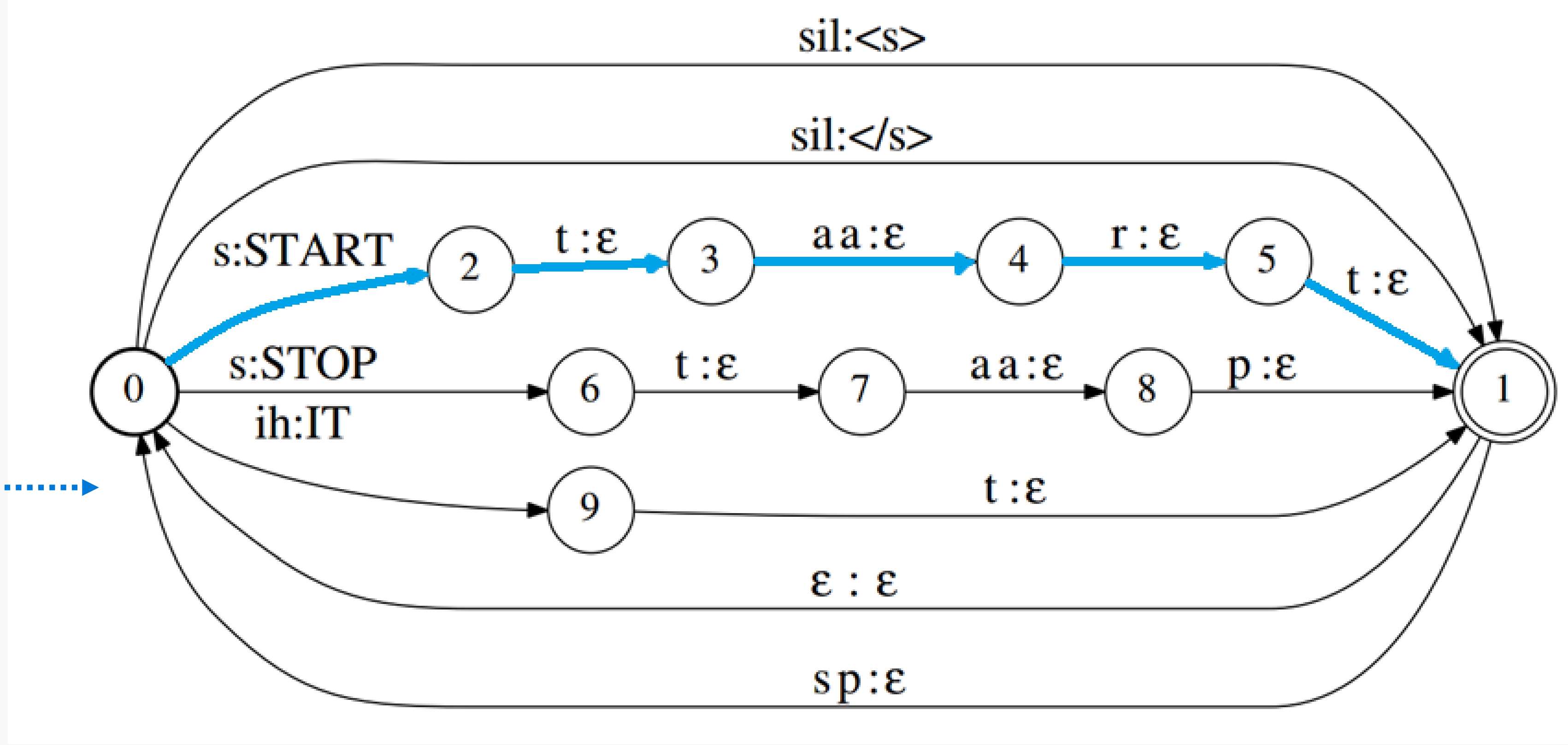


Introduction

- Weighted finite state transducer (WFST) Decoding



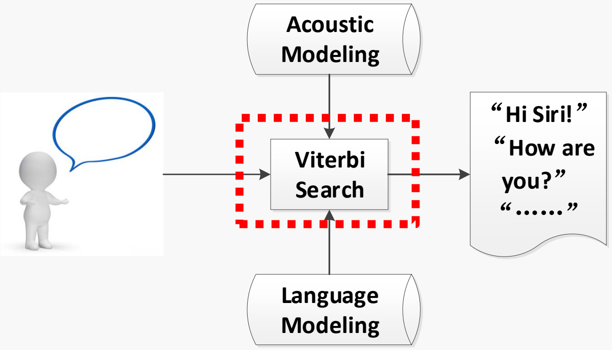
word	Pronunciation	Score[1]
<s>	sil	0.1
</s>	sil	0.6
START	s t aa r t	0.5
STOP	s t aa p	0.4
IT	ih t	0.3



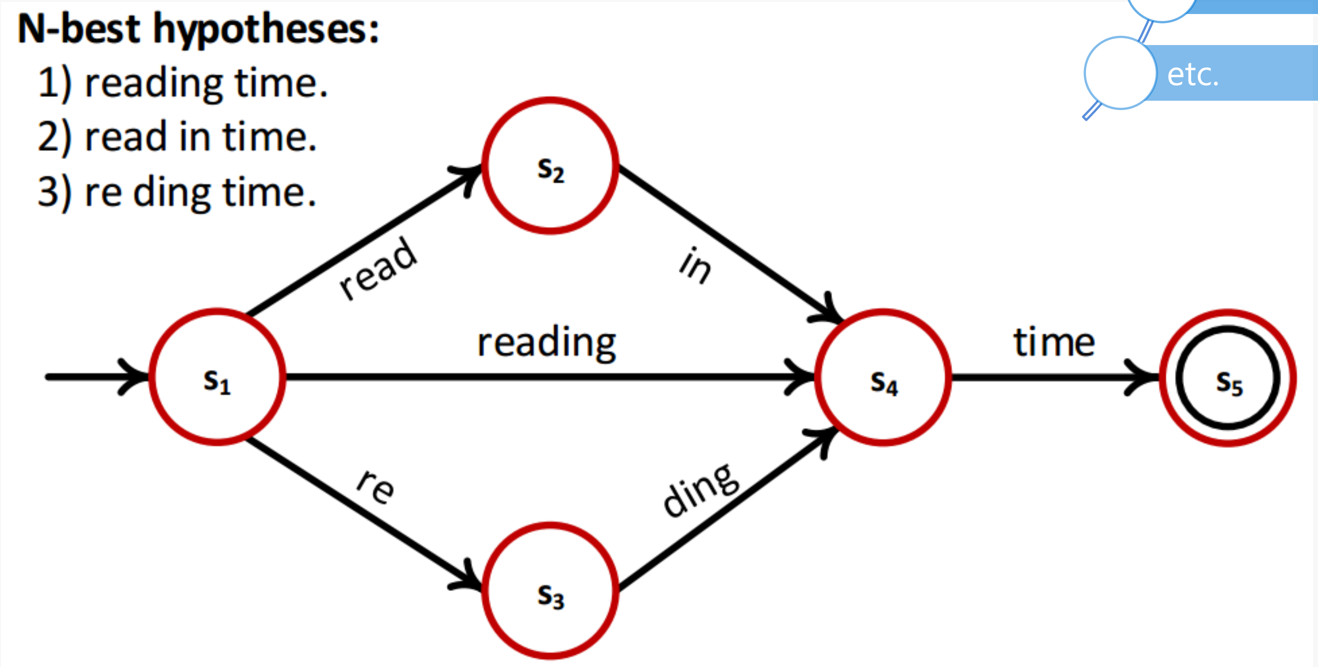
1. There is a score for each arc. For simplicity, we do not draw them.

Introduction

- Lattice Processing

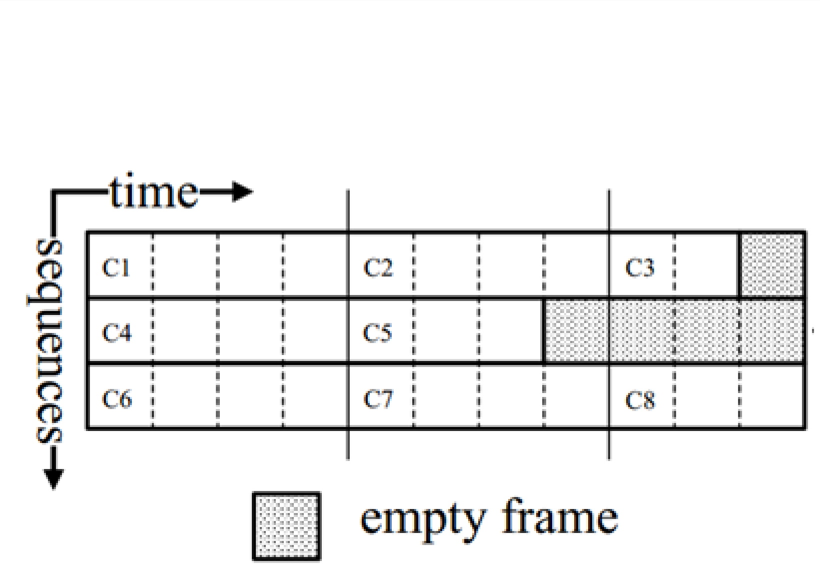


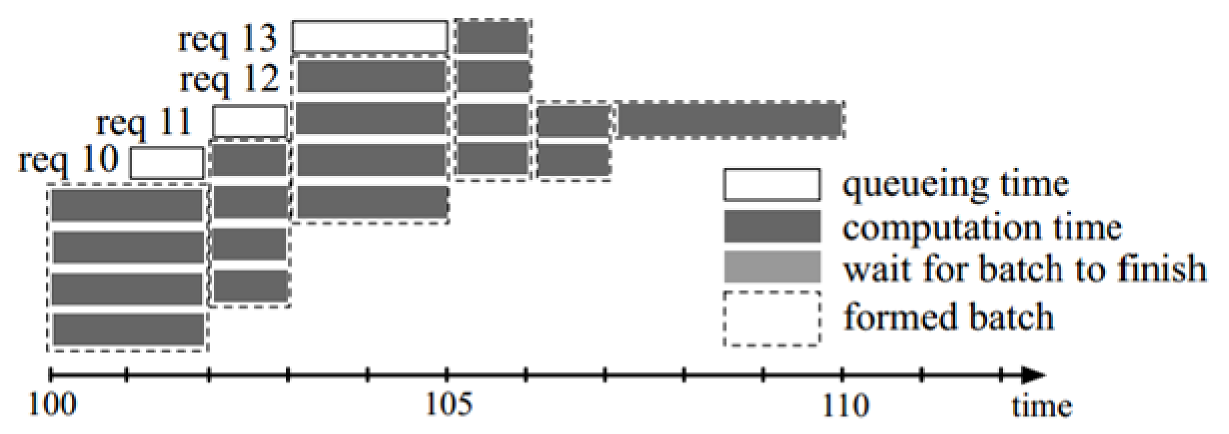
- Lattice rescoring
- Discriminative training
- Confidence measure
- Confusion network
- etc.



Introduction

- GPU-based Parallel Computing
  - Matrix calculation
  - Sequence batching



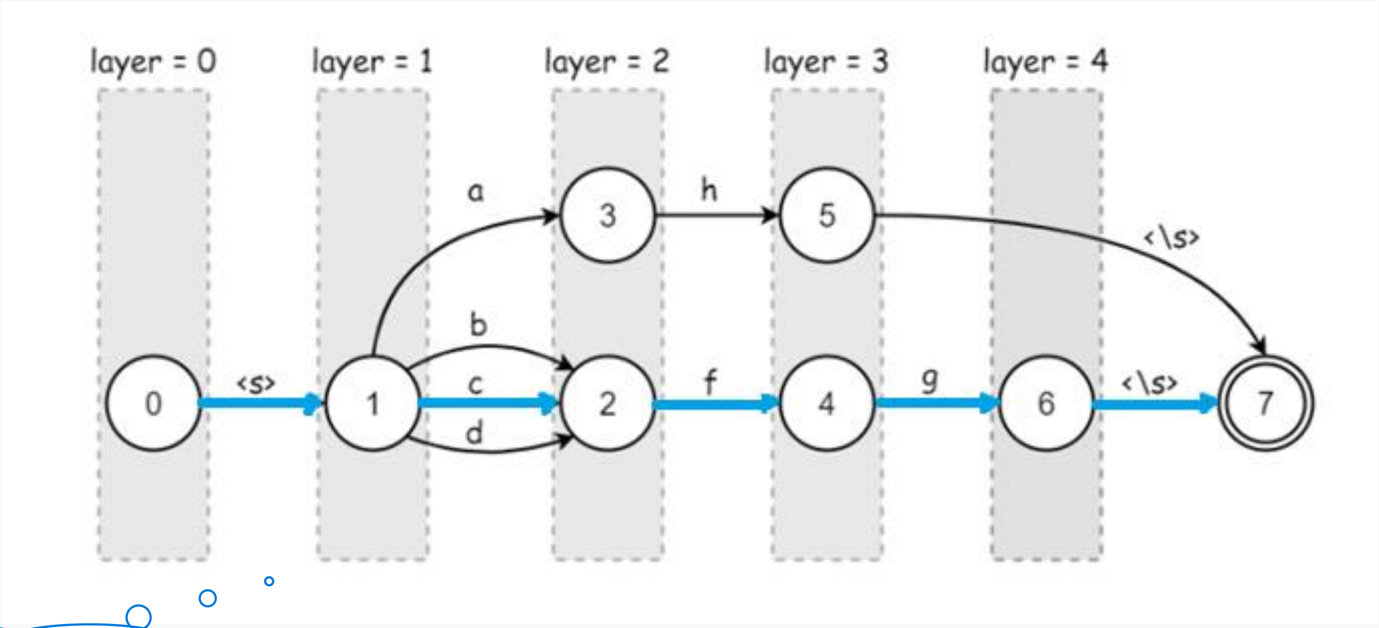


Training

Inference

Introduction

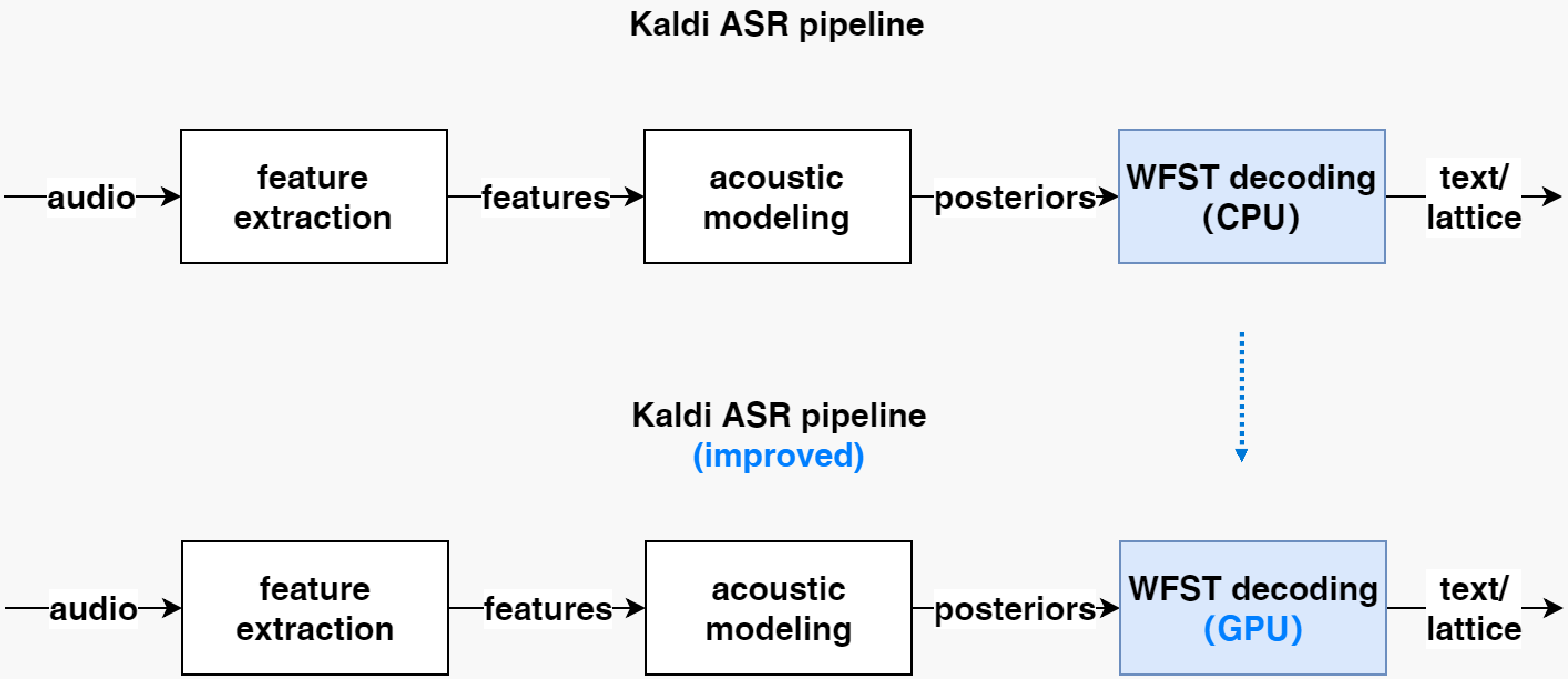
- This Work: **GPU-based Decoding**



Parallel Viterbi Search

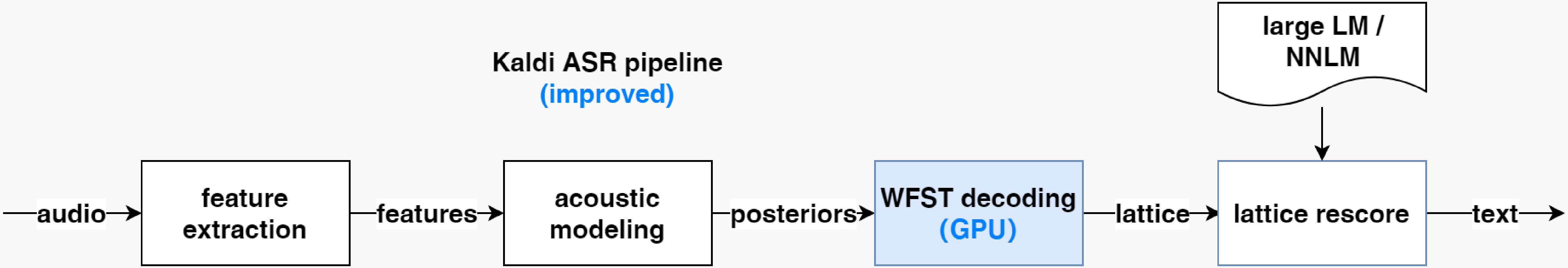
Introduction

- This Work: GPU-based Decoding



Introduction

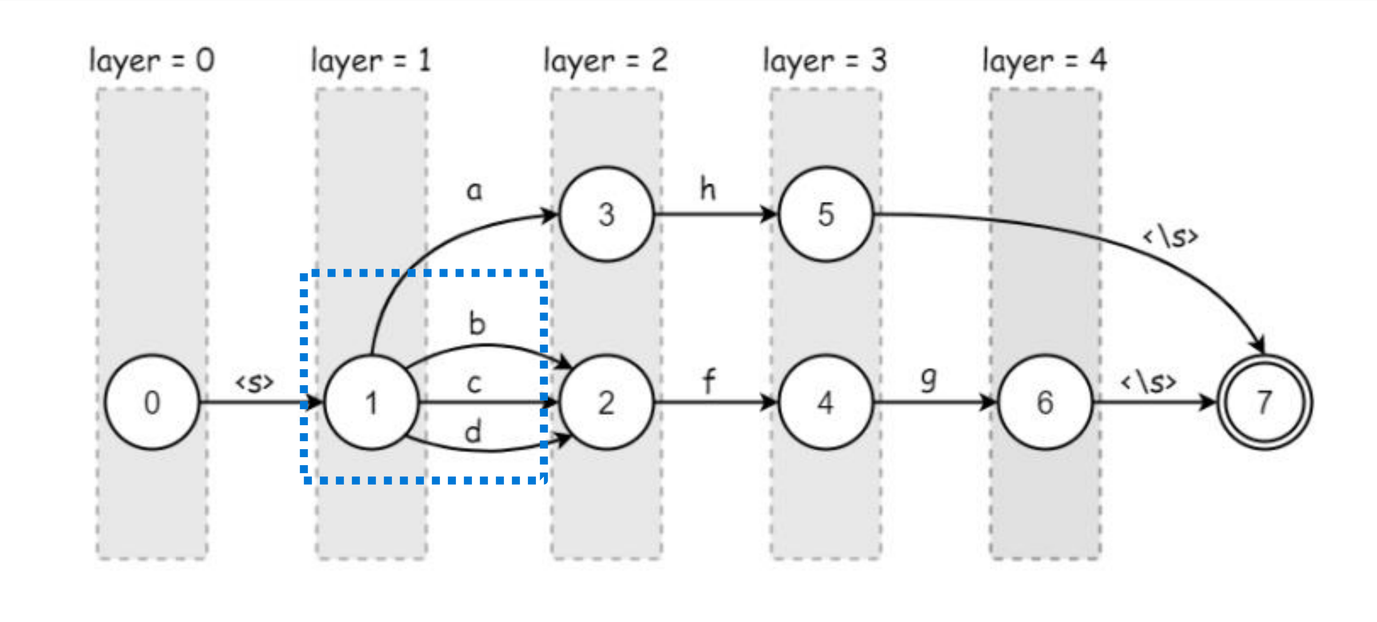
- This Work: GPU-based Decoding



  - GPU memory is enough for 1-pass WFST

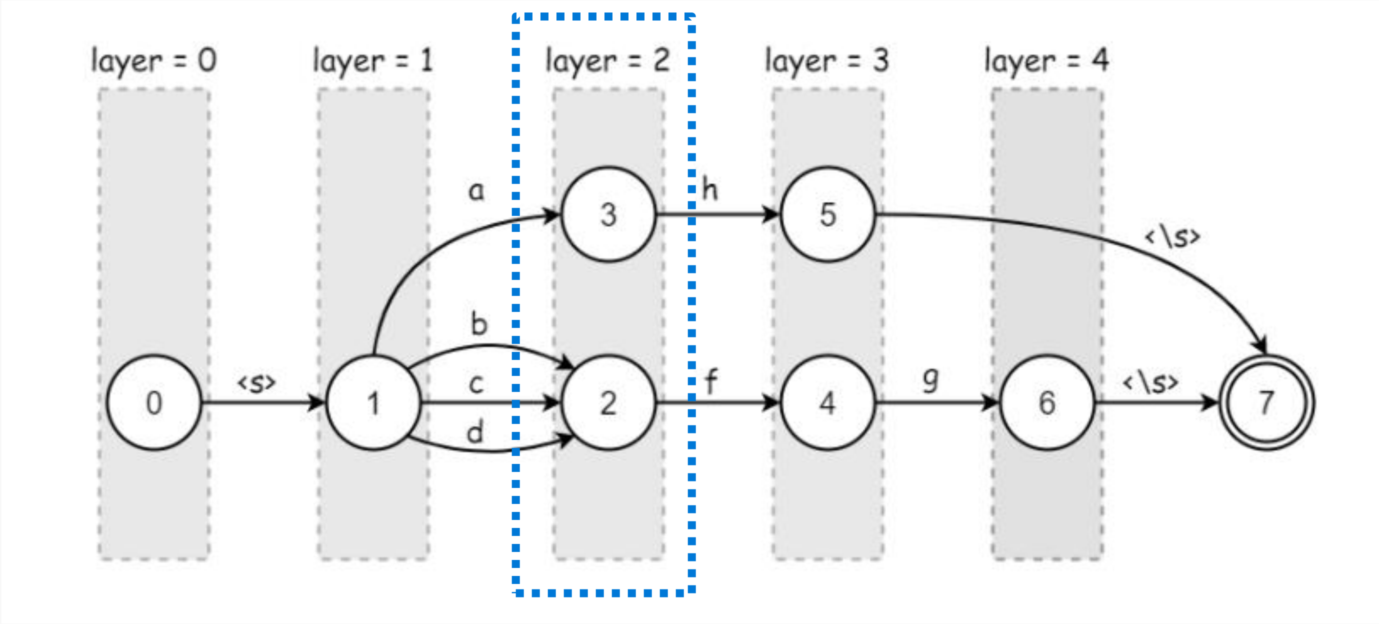
Framework

- **Parallel** Viterbi decoding
  - **Future:** out-going arcs, e.g. b, c & d



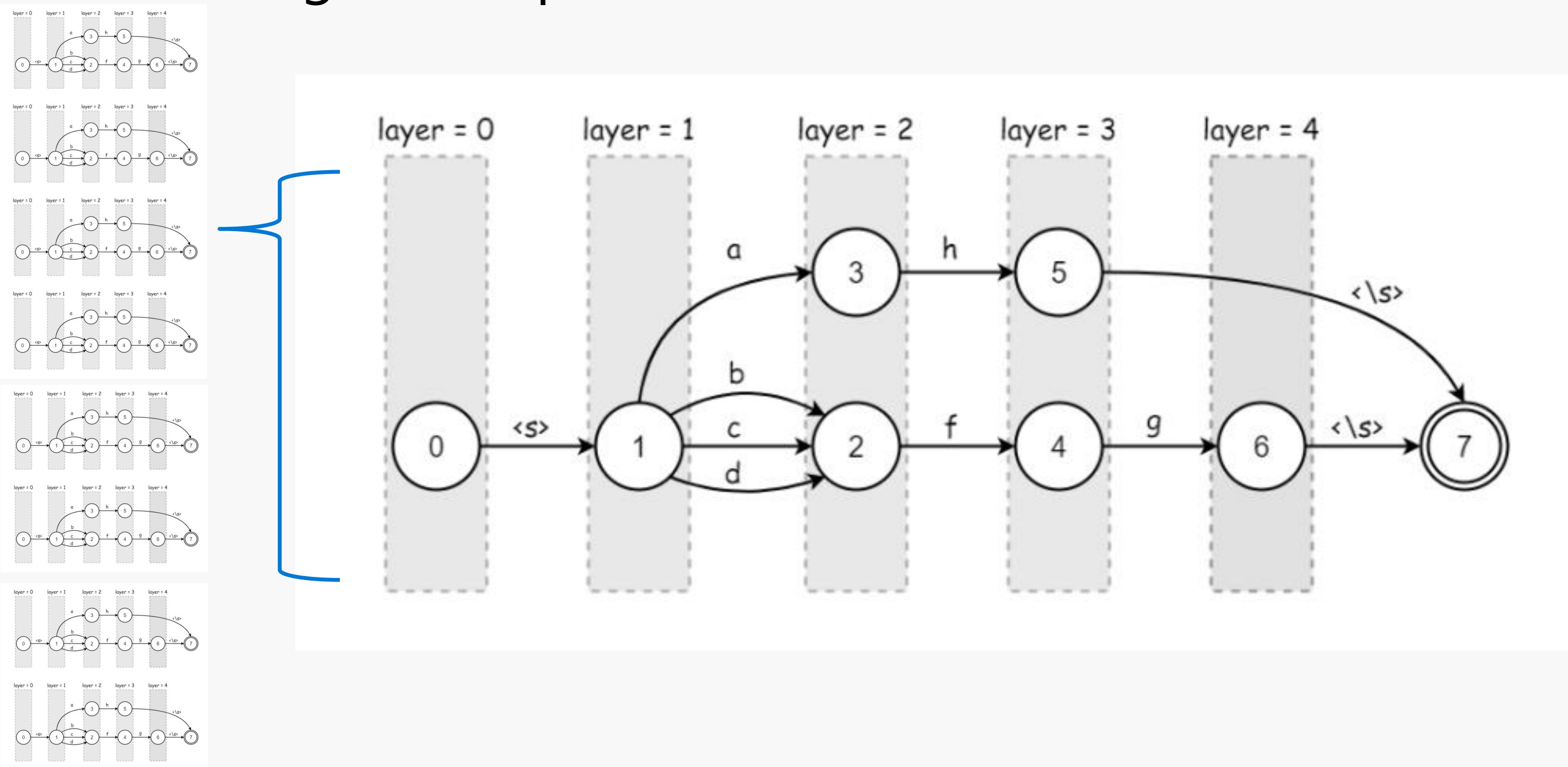
Framework

- Parallel Viterbi decoding
  - **Future:** out-going arcs, e.g. b, c & d
  - **History:** per layer, e.g. 3 & 2



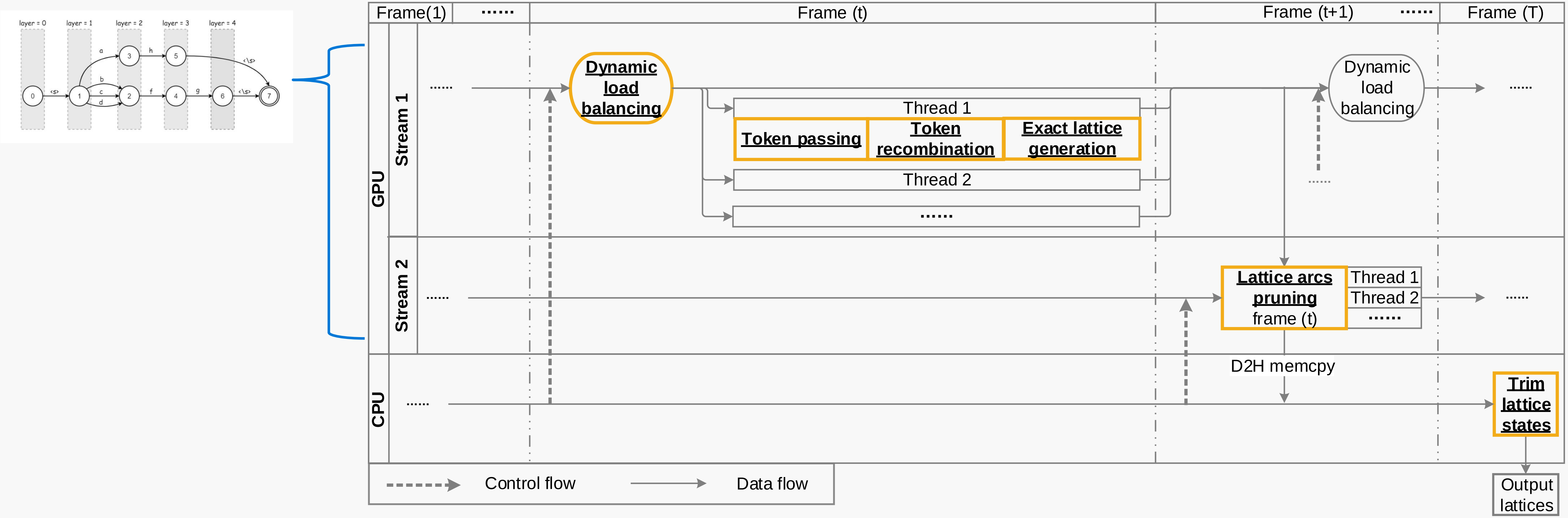
Framework

- **Parallel** Viterbi decoding
  - **Future:** out-going arcs, e.g. b, c & d
  - **History:** per layer, e.g. 3 & 2
  - **Utterance:** decoding with separate GPU kernels



Framework

- System overview
  - 2 GPU streams & 1 CPU thread



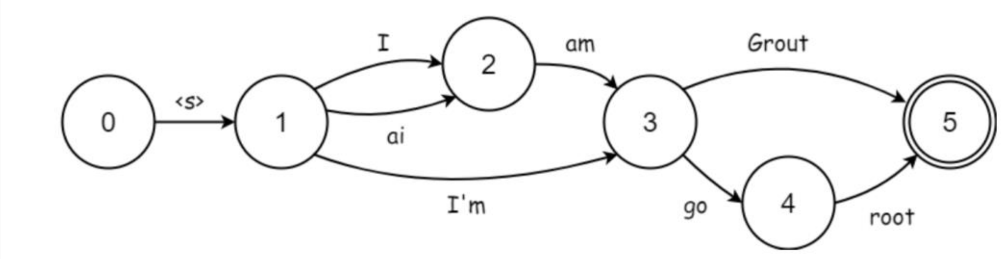
1st Problem: Token recombination

- Token recombination

Language Model **Dictionary** **DNN-HMM**

$$\max P(\mathbf{W}|\mathbf{O}) \propto \max P(\mathbf{W}) \cdot P(\mathbf{L}|\mathbf{W}) \cdot \sum_{\mathbf{q}\in\mathcal{A}(\mathbf{L})} p(\mathbf{O},\mathbf{q}|\mathbf{L}) \qquad (1)$$

$$\propto \max P(\mathbf{W}) \cdot P(\mathbf{L}|\mathbf{W}) \cdot \max_{\mathbf{q}\in\mathcal{A}(\mathbf{L})} p(\mathbf{O},\mathbf{q}|\mathbf{L}) \qquad (2)$$



Token recombination

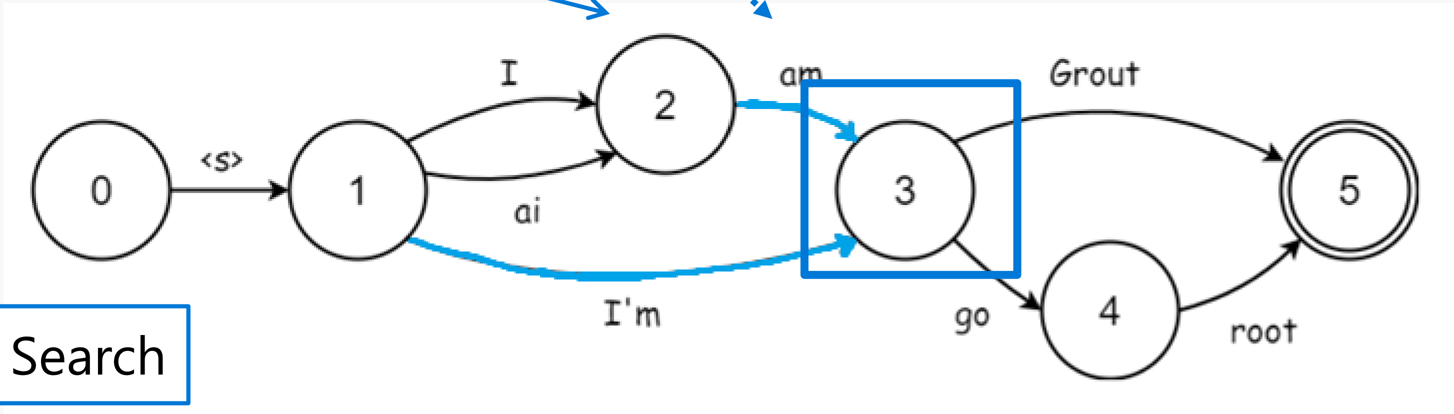
- Token recombination

$$\max P(\mathbf{W}|\mathbf{O}) \propto \boxed{\max} P(\mathbf{W}) \cdot P(\mathbf{L}|\mathbf{W}) \cdot \boxed{\max_{\mathbf{q}\in\mathcal{A}(\mathbf{L})}} p(\mathbf{O},\mathbf{q}|\mathbf{L}) \qquad (2)$$

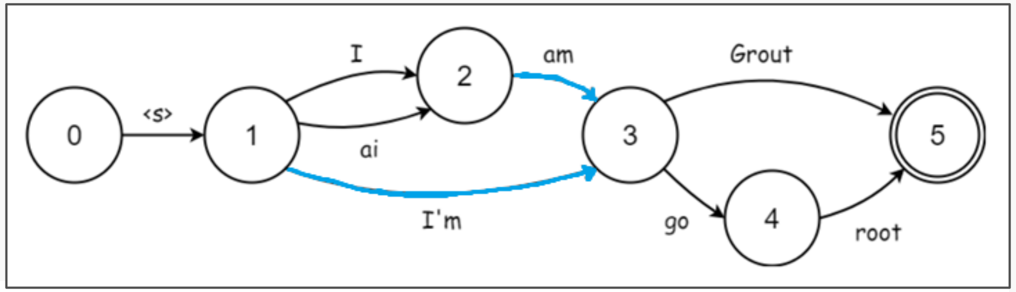
Serial Viterbi Search

We need a **Thread Sync.**

Parallel Viterbi Search



Token recombination



- Naïve implementation:
  - Critical section

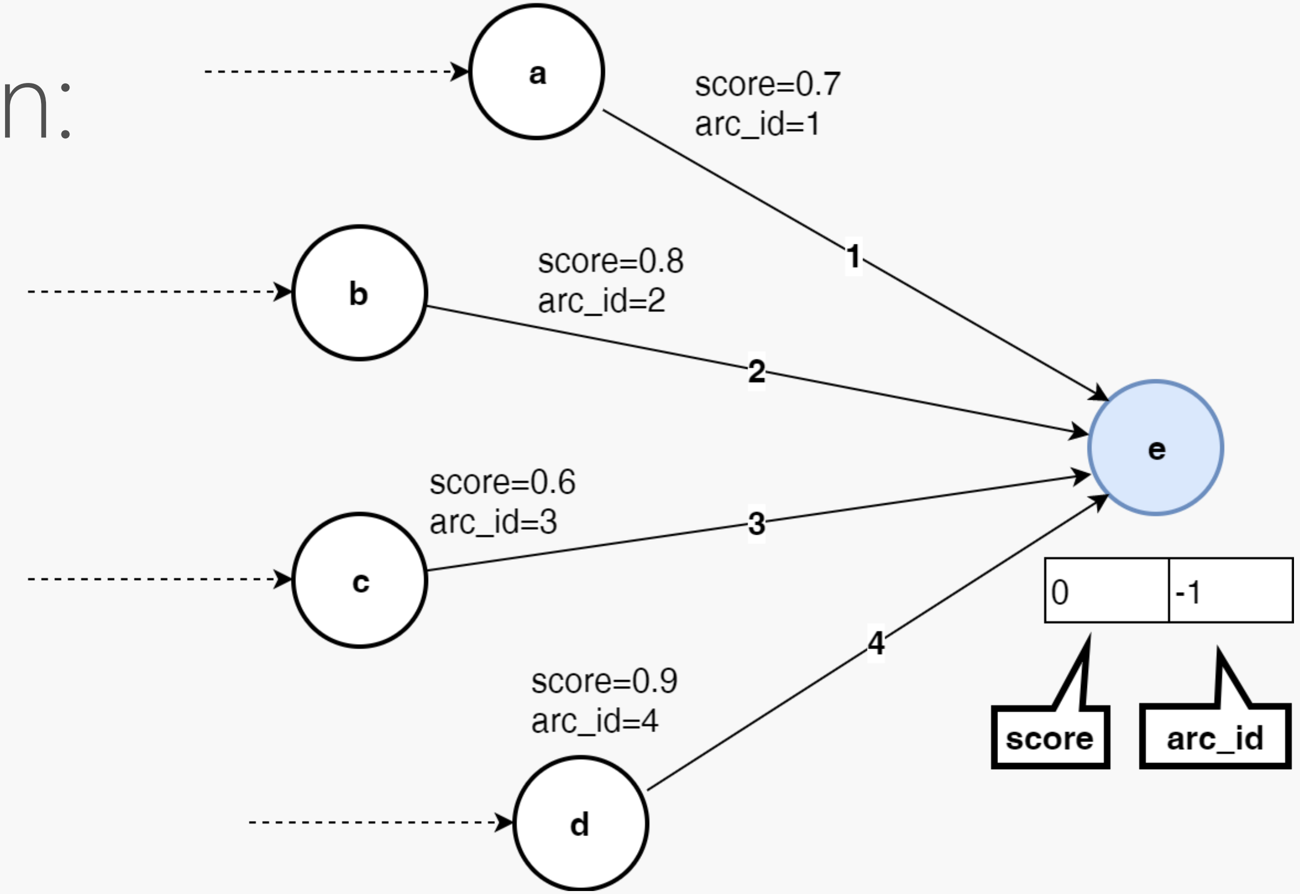
```
while(*cur_tokv < next_tok) {   // check if we need to recombine token
    acquire_semaphore((int*)&params.token_locks[nextstate]); // per token lock
    if(*cur_tokv < next_tok) { // double check if we are min
        *cur_tok=next_tok; // this is what we really want to do
    }
    release_semaphore((int*)&params.token_locks[nextstate]); // release per token lock
    break; // exit loop as our update is done
} // end while
```

- Thread deadlock in earlier GPU architectures
- Slow because of **while loop** and **semaphore acquiring**

Token recombination

- Proposed implementation:
  - Single atomic operation[1]

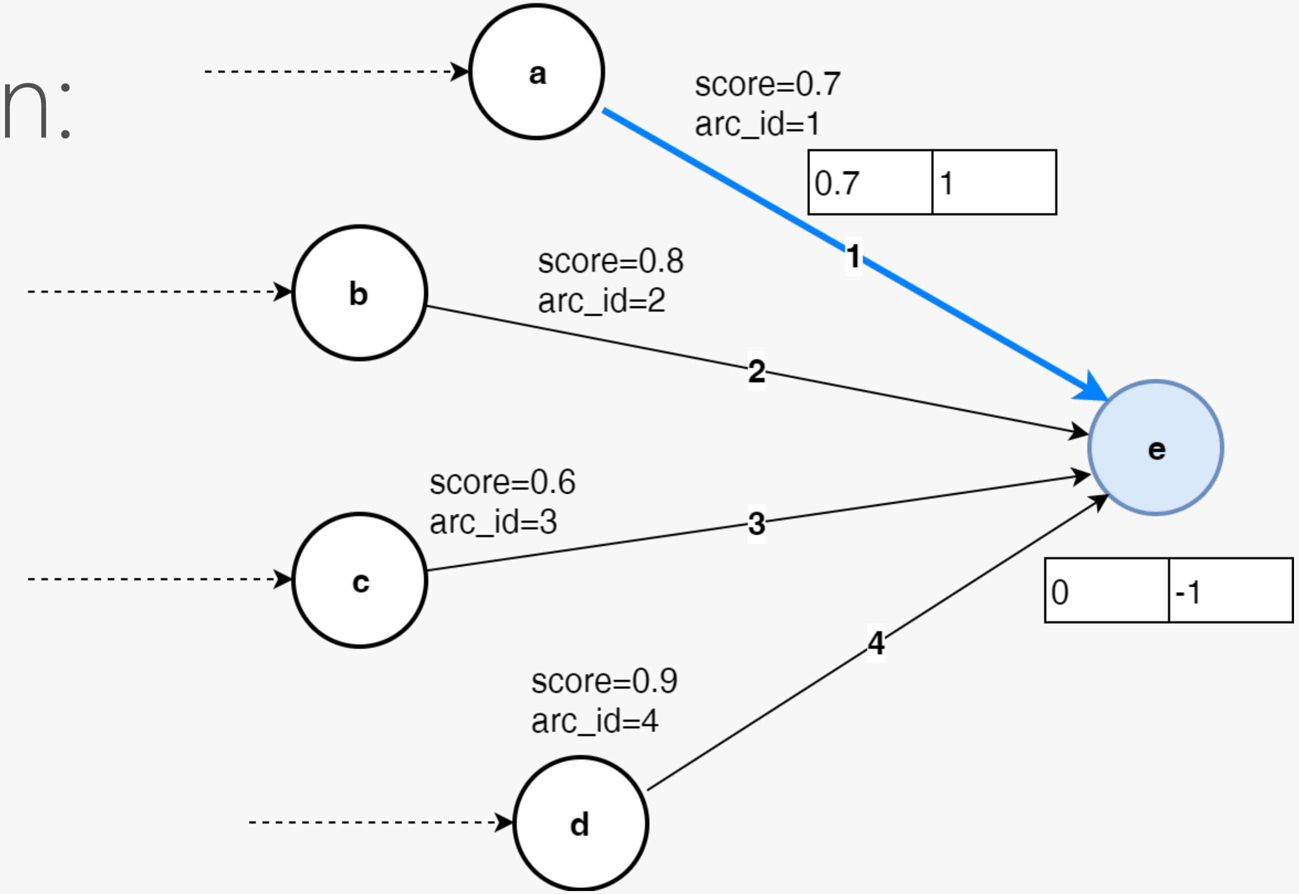
```
int64* atomicMax(*address, val) {
    original = *address;
    *address = max(val, *address);
    return original;
}
```



1. *atomicMax*(**address*, *val*). Computes *max*(**address*, *val*), writes the result to *address*, and returns the original **address* .

Token recombination

- Proposed implementation:
  - Single atomic operation[1]

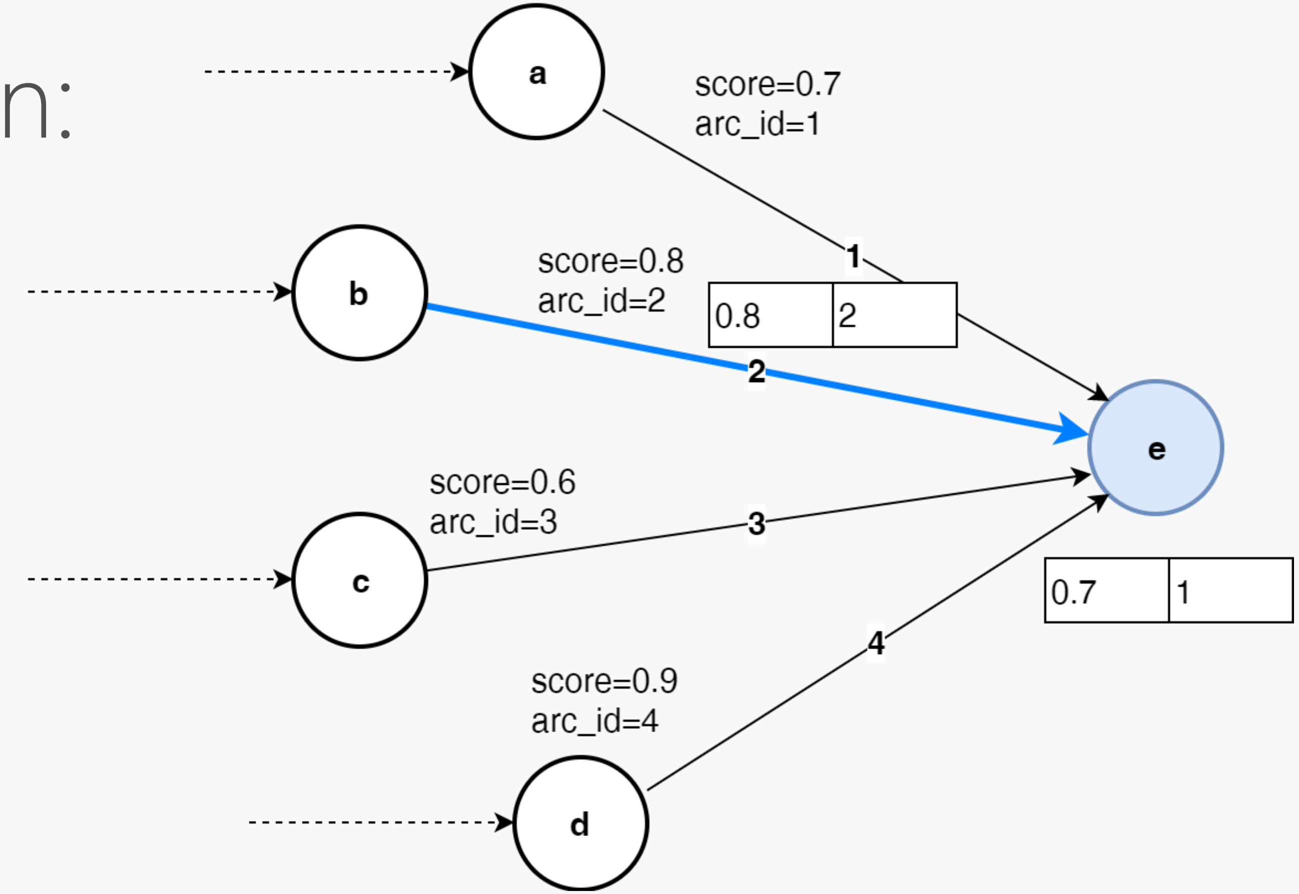



```
int64* atomicMax(*address, val) {
    original = *address;
    *address = max(val, *address);
    return original;
}
```

1. *atomicMax*(**address*, *val*). Computes *max*(**address*, *val*), writes the result to *address*, and returns the original **address* .

Token recombination

- Proposed implementation:
  - Single atomic operation[1]



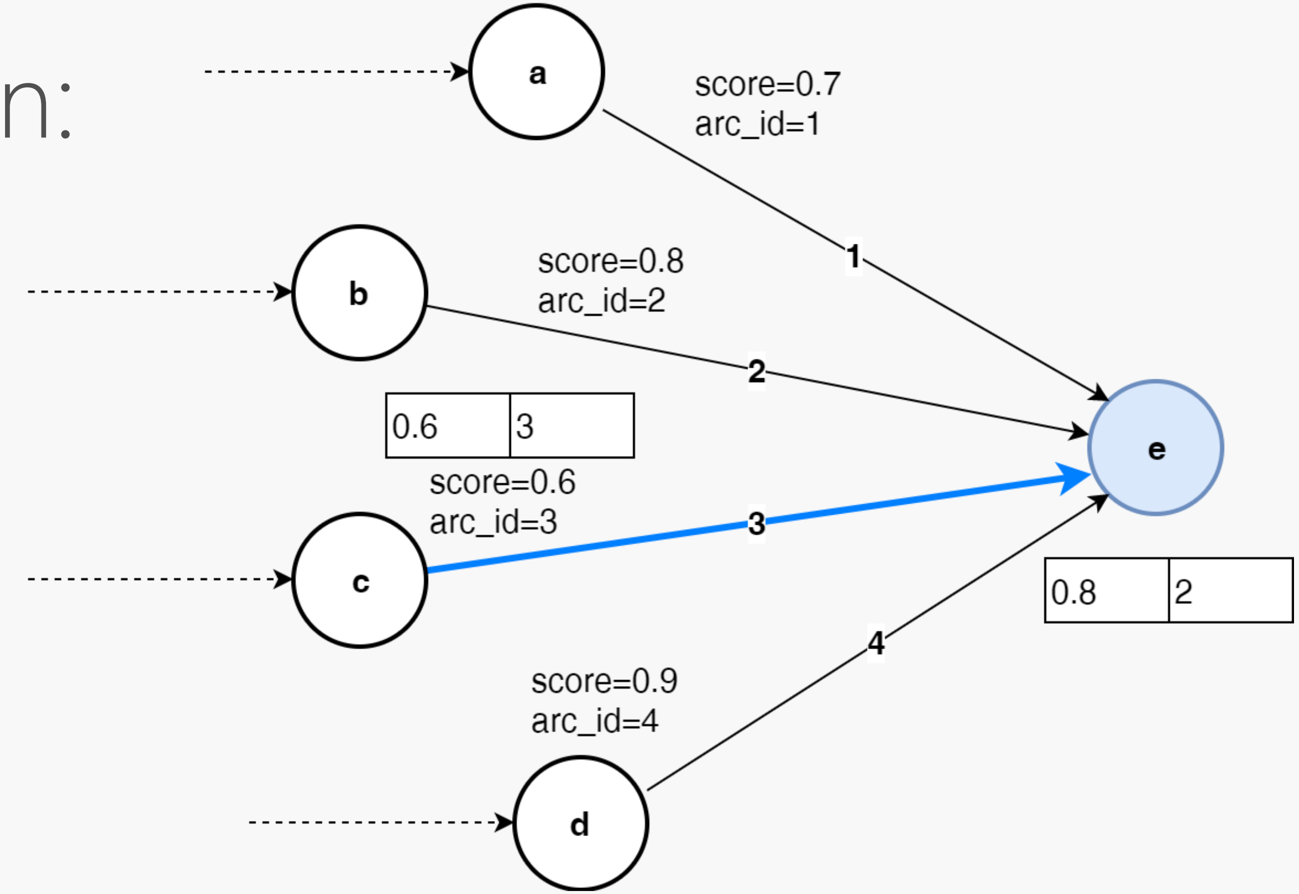

```
int64* atomicMax(*address, val) {
    original = *address;
    *address = max(val, *address);
    return original;
}
```

1. *atomicMax*(**address*, *val*). Computes *max*(**address*, *val*), writes the result to *address*, and returns the original **address* .

Token recombination

- Proposed implementation:
  - Single atomic operation[1]

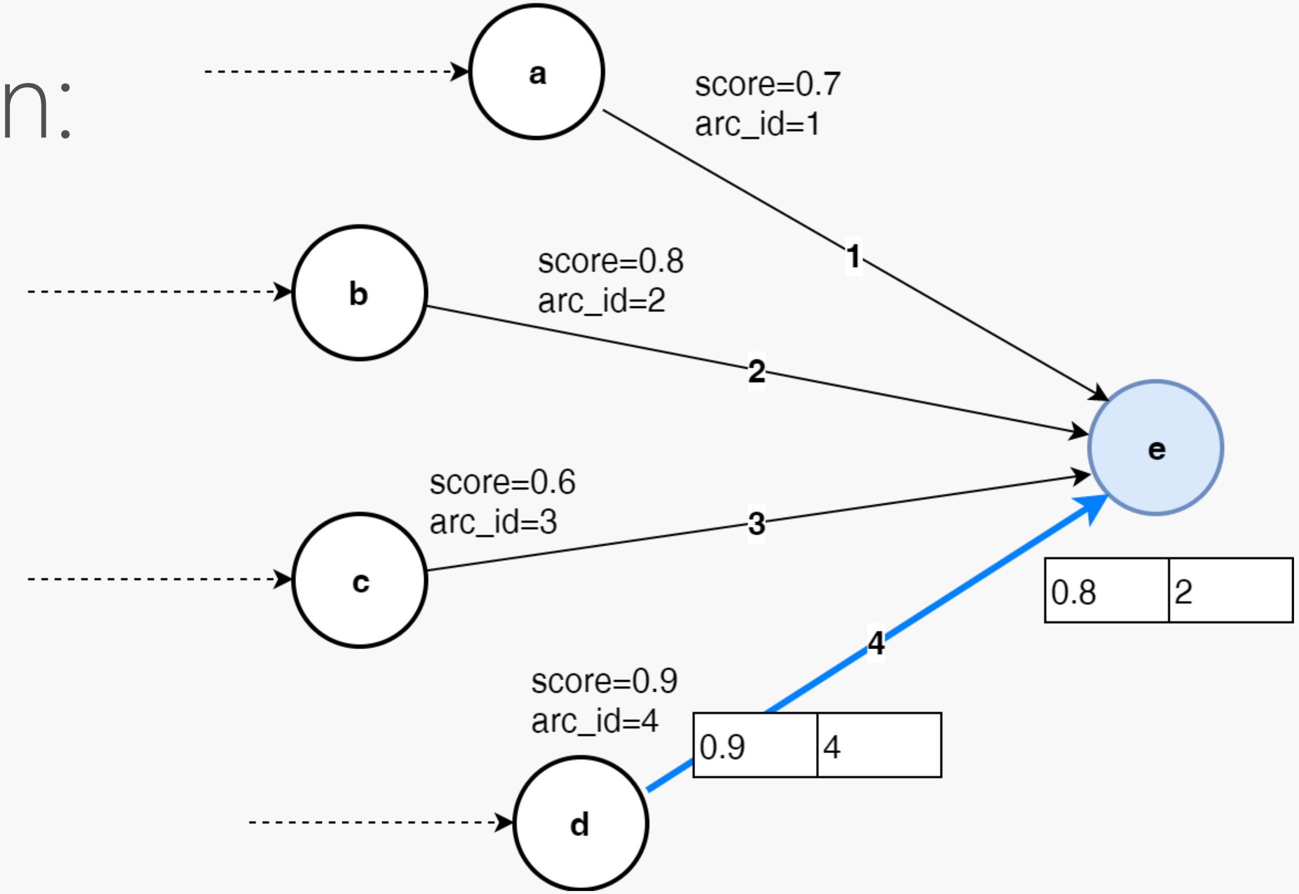
```
int64* atomicMax(*address, val) {
    original = *address;
    *address = max(val, *address);
    return original;
}
```



1. *atomicMax*(**address*, *val*). Computes *max*(**address*, *val*), writes the result to *address*, and returns the original **address* .

Token recombination

- Proposed implementation:
  - Single atomic operation[1]




```
int64* atomicMax(*address, val) {
    original = *address;
    *address = max(val, *address);
    return original;
}
```

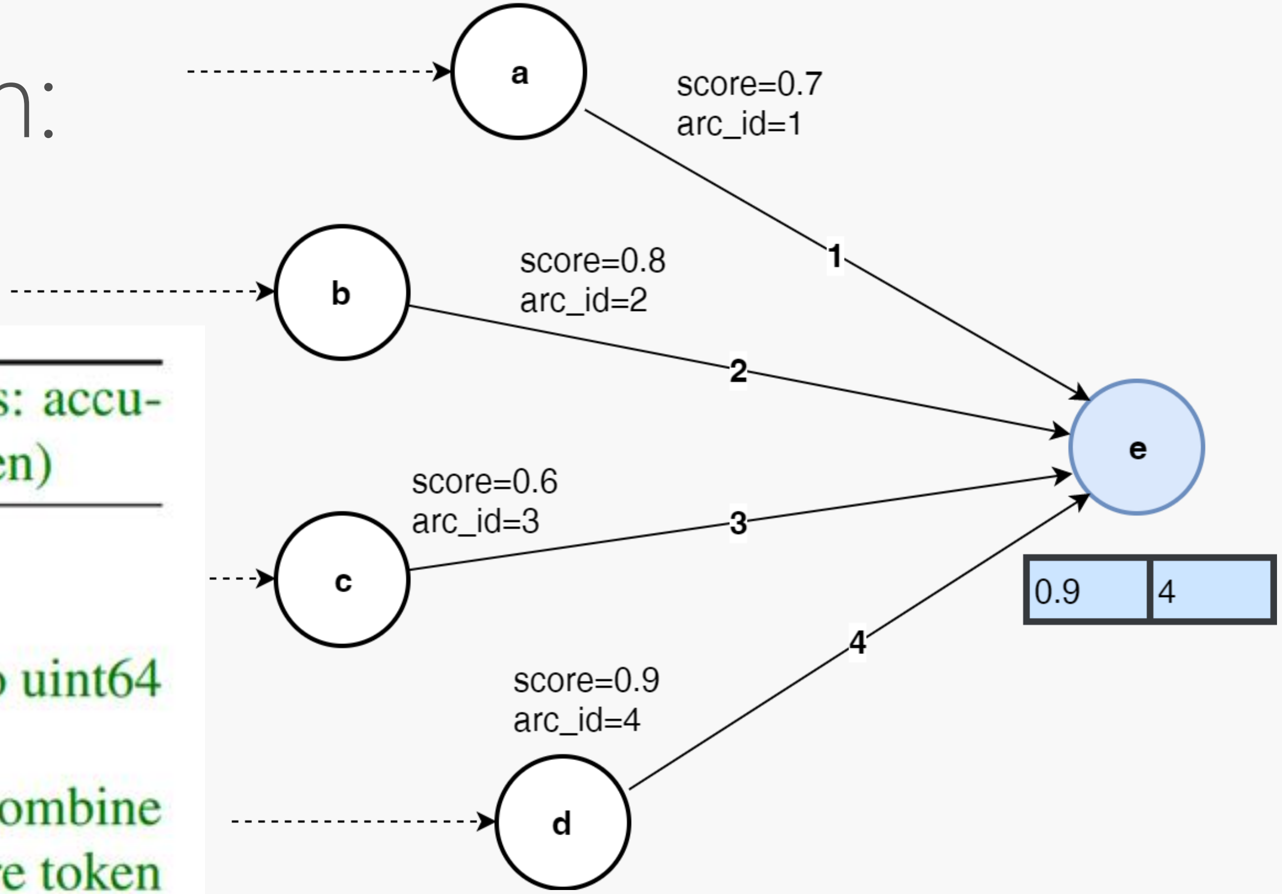
1. *atomicMax*(**address*, *val*). Computes *max*(**address*, *val*), writes the result to *address*, and returns the original **address* .

Token recombination

- Proposed implementation:
  - Single atomic operation[1]

Algorithm 1 Thread-level Token Recombination (Inputs: accumulated cost, an out-going WFST arc and a current token)

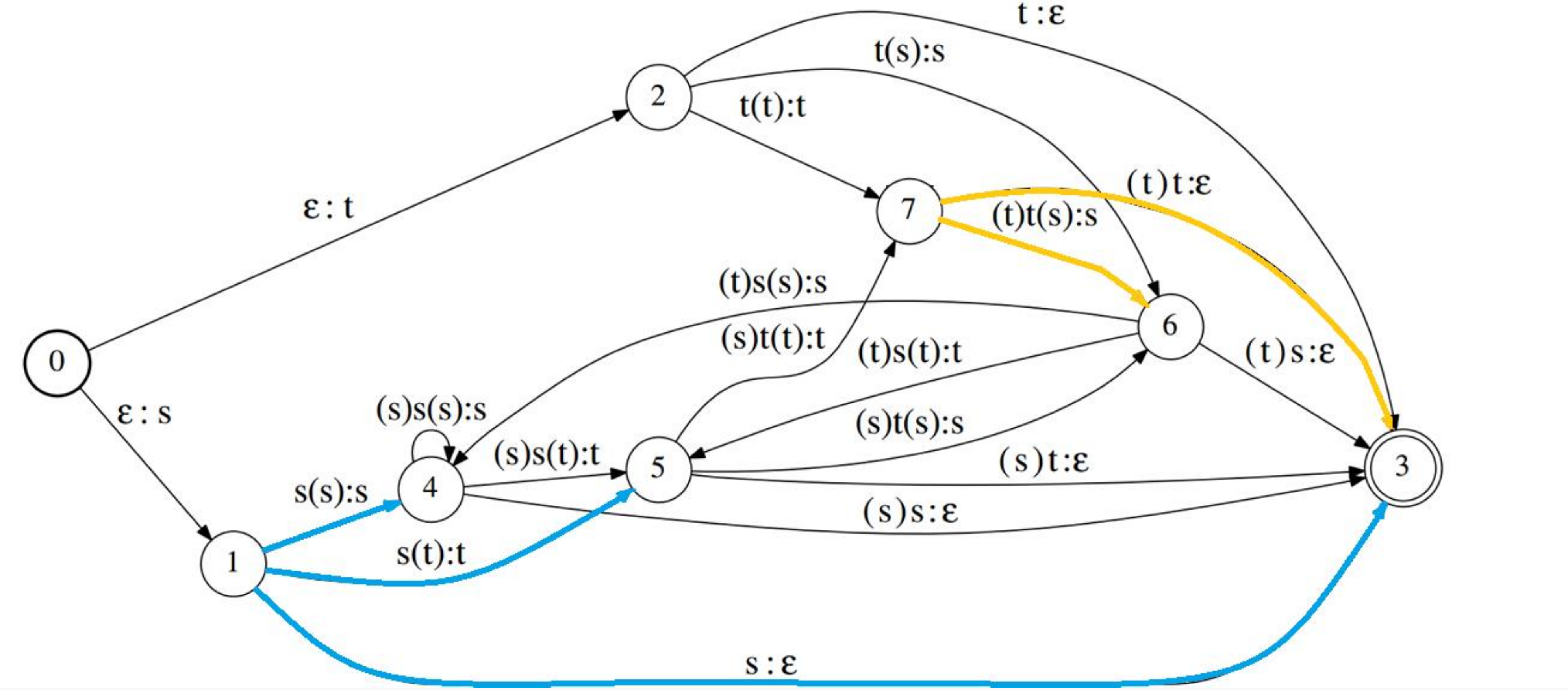
```
procedure RECOMBINE(cost, arc, curTok)
    oldTokPack = state2tokPack[arc.next_state]
    curTokPack = packFunc(cost,arc.id)  ▷ pack into uint64
    ret = atomicMin (oldTokPack,curTokPack)
    if ret > curTokPack then                ▷ recombine
        perArcTokBuf[arc.id] = *curTok      ▷ store token
```



1. *atomicMax*(**address*, *val*). Computes *max*(**address*, *val*), writes the result to *address*, and returns the original **address* .

2nd Problem: Load balancing

- Load imbalance: different num. of out-going arcs

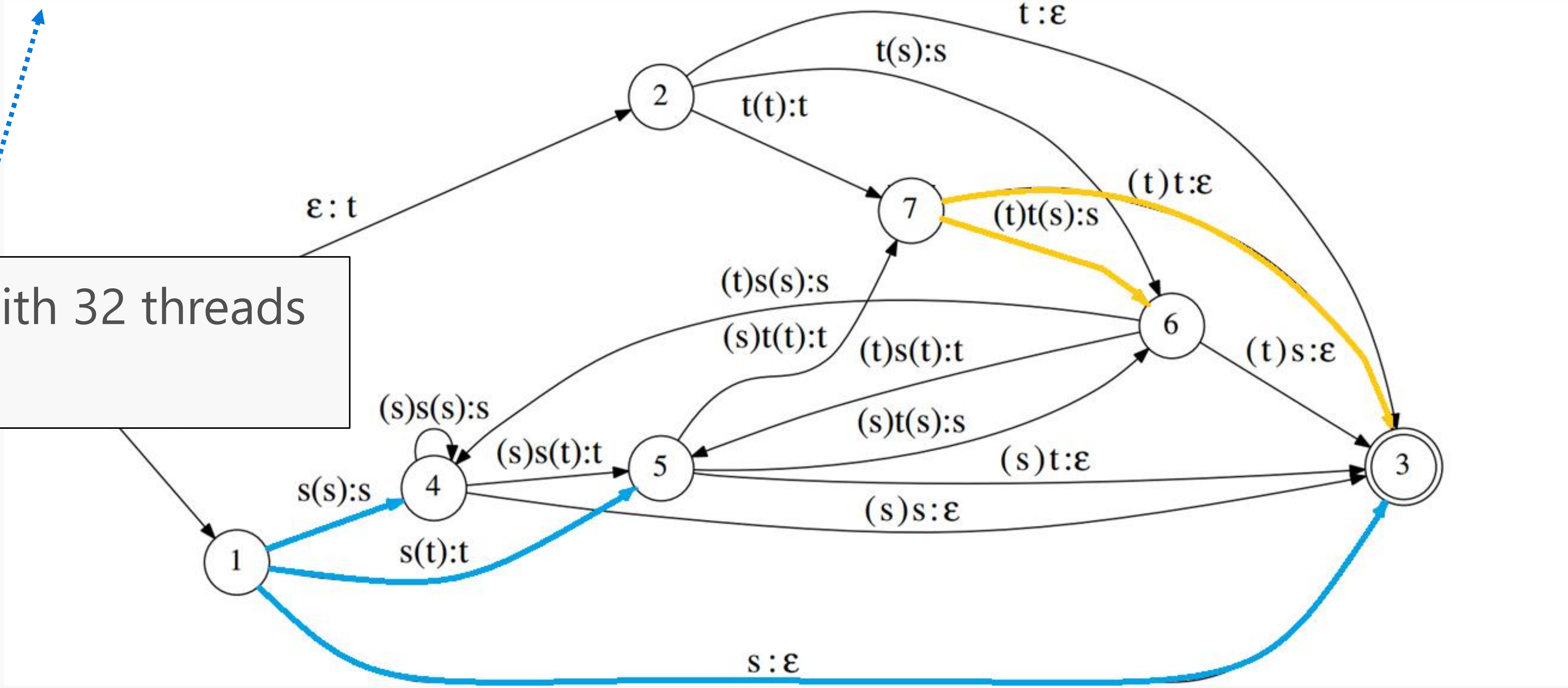


Load balancing

Group 0 with 32 threads
Token ?

Dispatcher
`idx=atomicAdd(global_d,1)`

Group 1 with 32 threads
Token ?

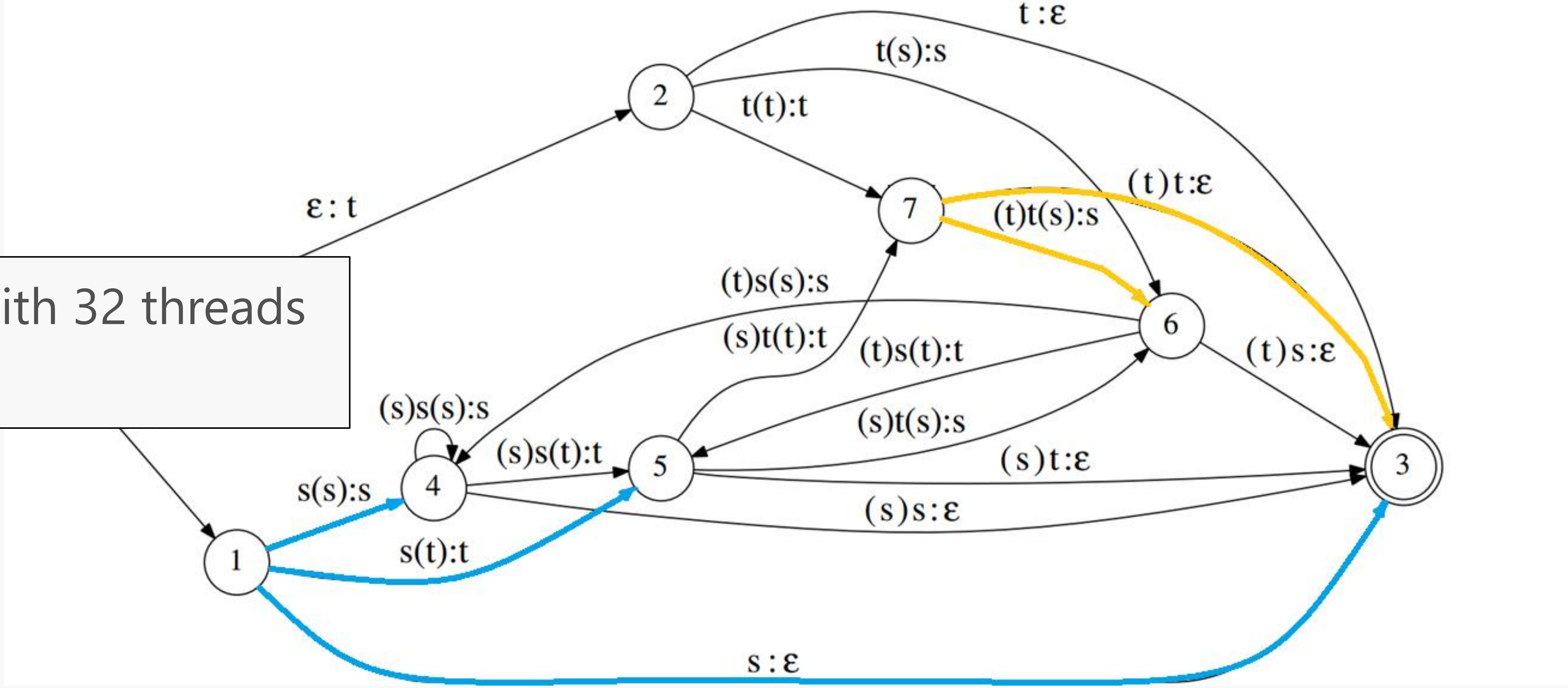


Load balancing

Group 0 with 32 threads
Token 0 in State 2

Dispatcher
idx=atomicAdd(global_d,1)

Group 1 with 32 threads
Token ?

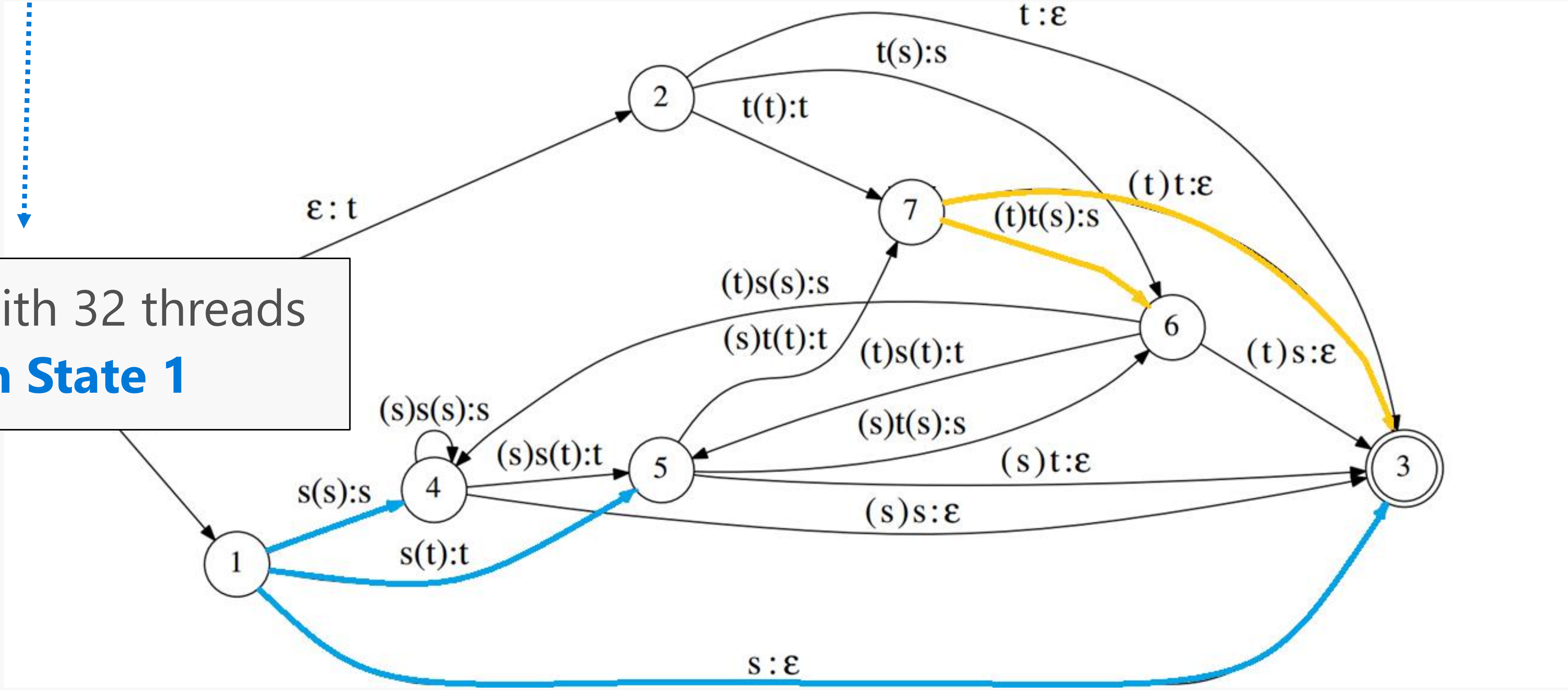


Load balancing

Group 0 with 32 threads
Token 0 in State 2

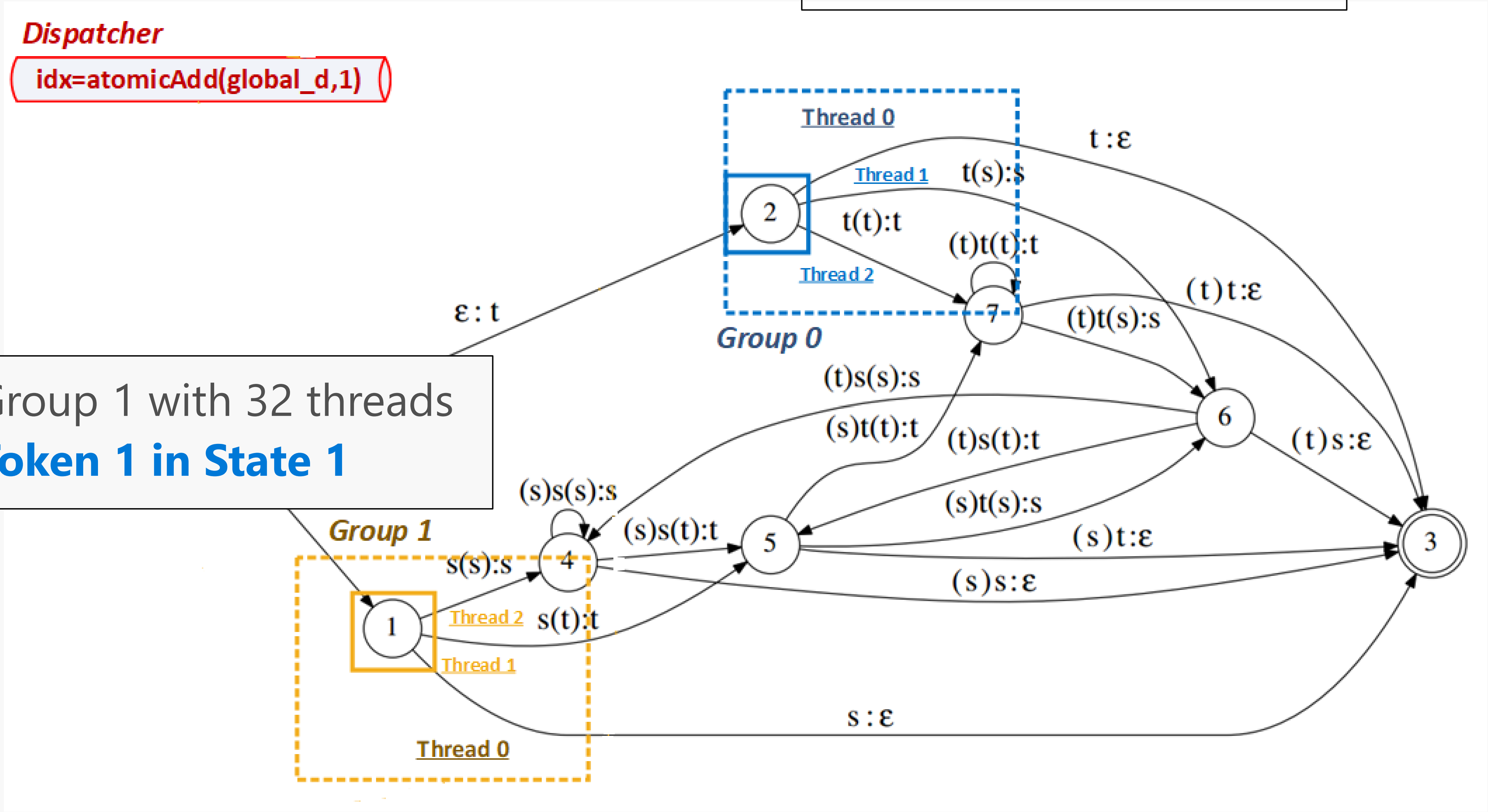
Dispatcher
idx=atomicAdd(global_d,1)

Group 1 with 32 threads
Token 1 in State 1



Load balancing

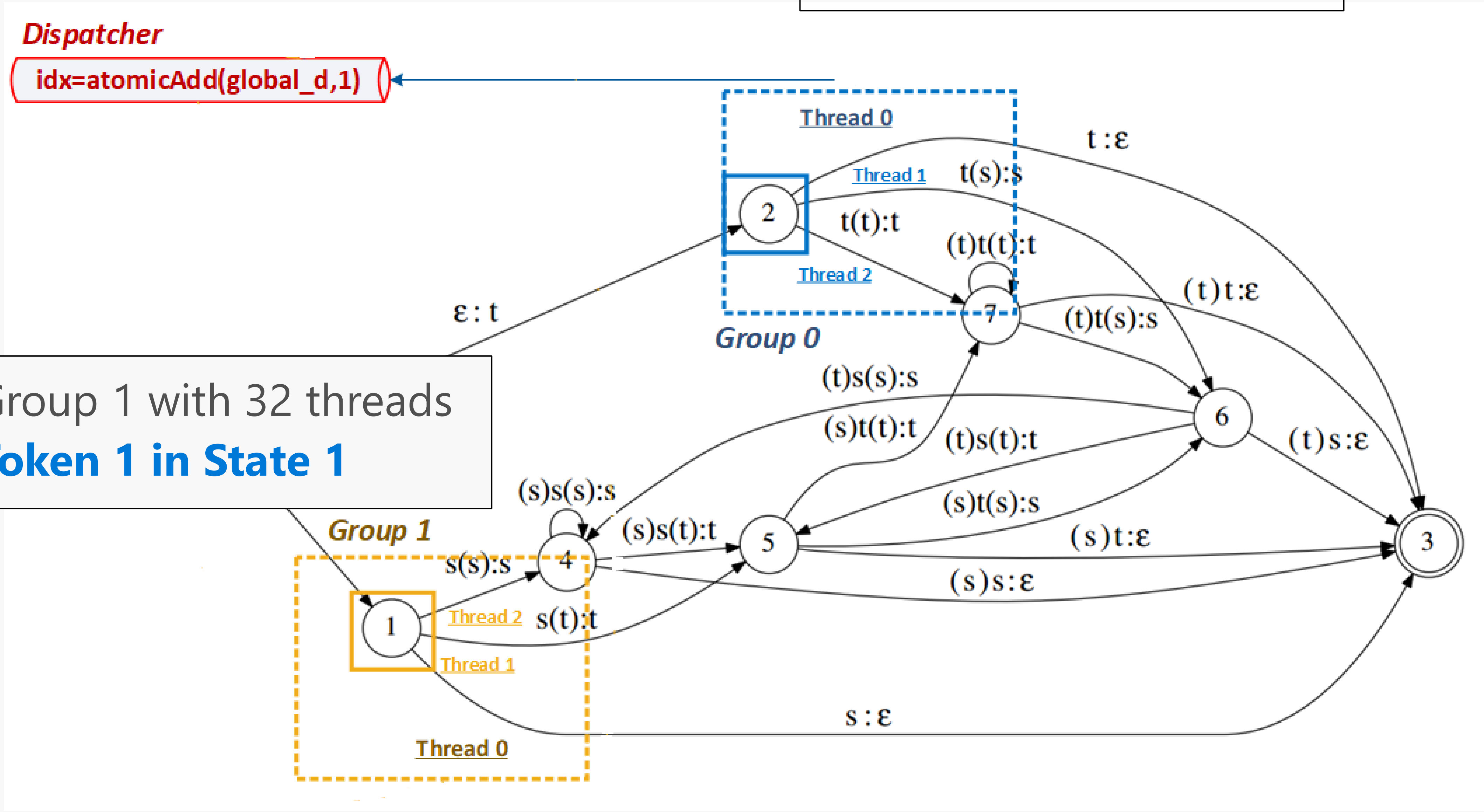
Group 0 with 32 threads
Token 0 in State 2



Group 1 with 32 threads
Token 1 in State 1

Load balancing

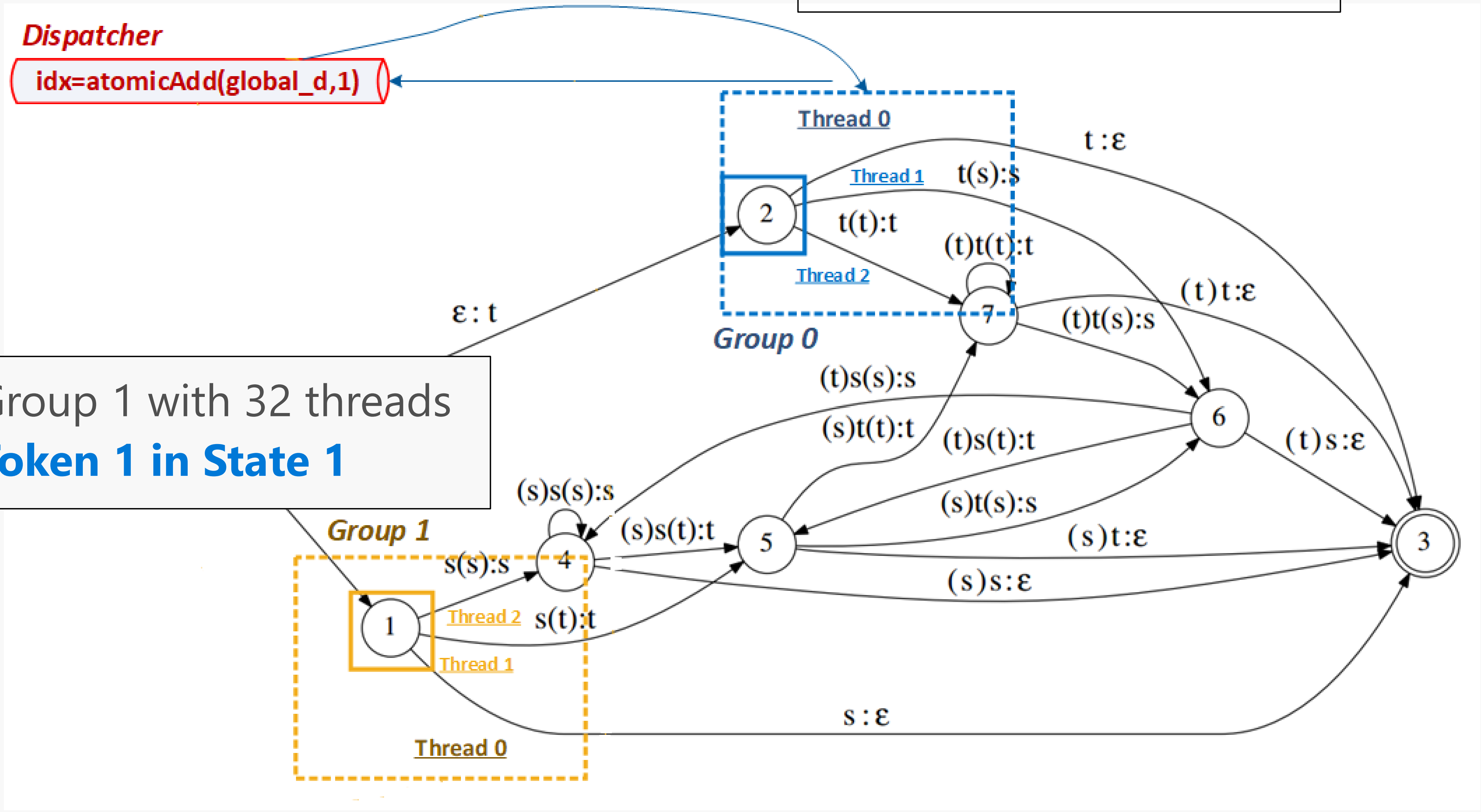
Group 0 with 32 threads
Token ?



Group 1 with 32 threads
Token 1 in State 1

Load balancing

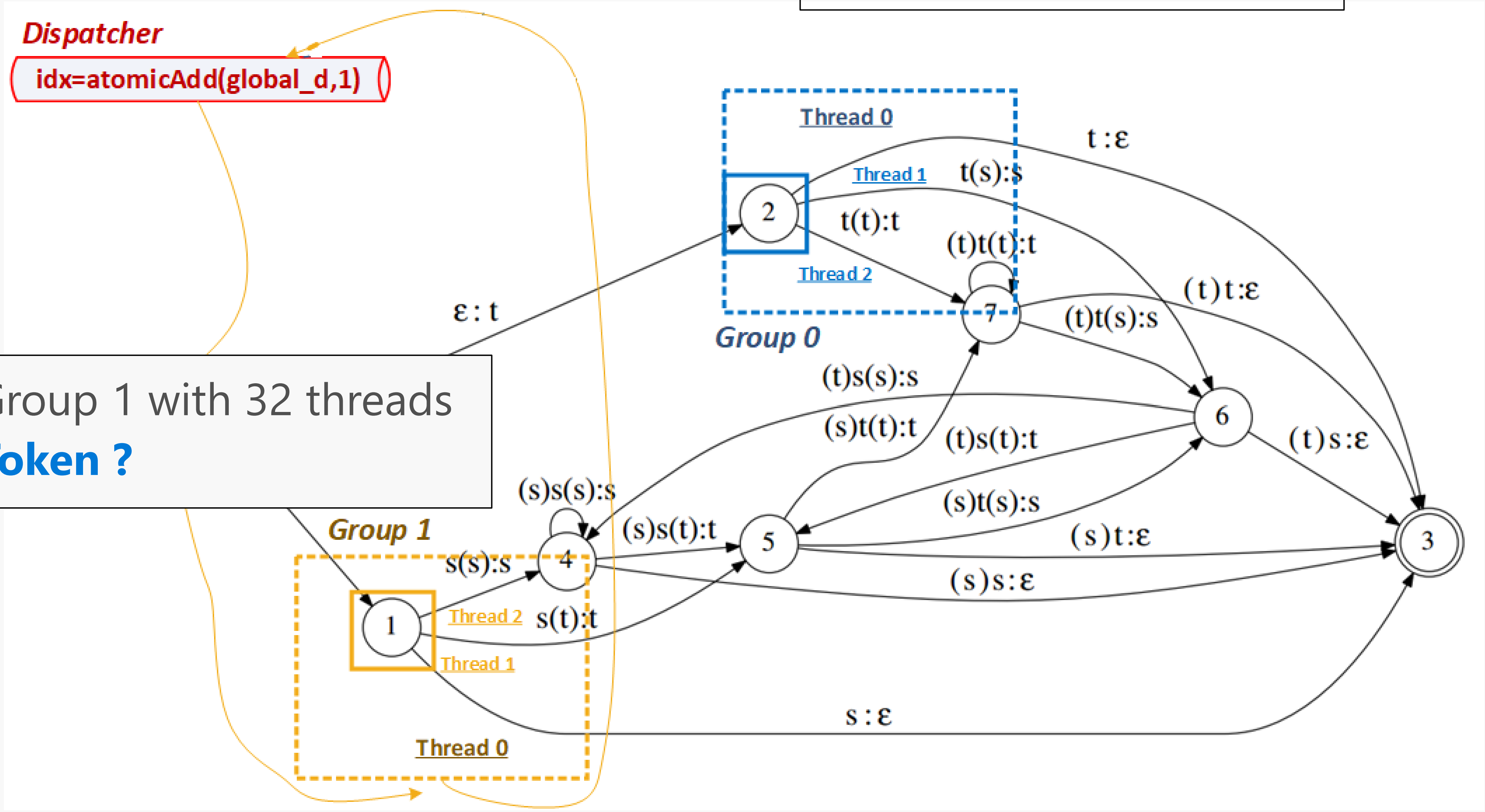
Group 0 with 32 threads
Token ?



Group 1 with 32 threads
Token 1 in State 1

Load balancing

Group 0 with 32 threads
Token 0 in State 2

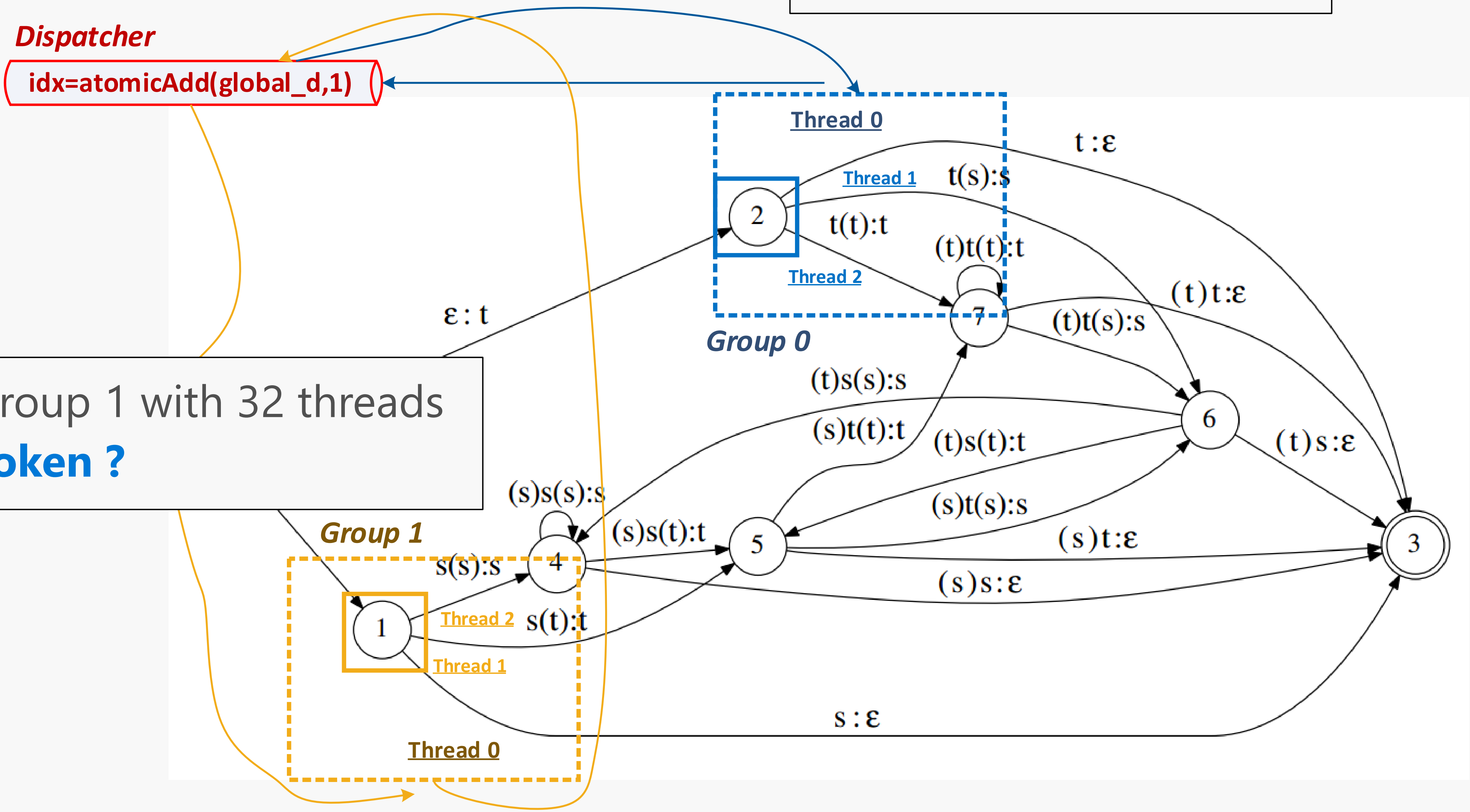


Group 1 with 32 threads
Token ?

Load balancing

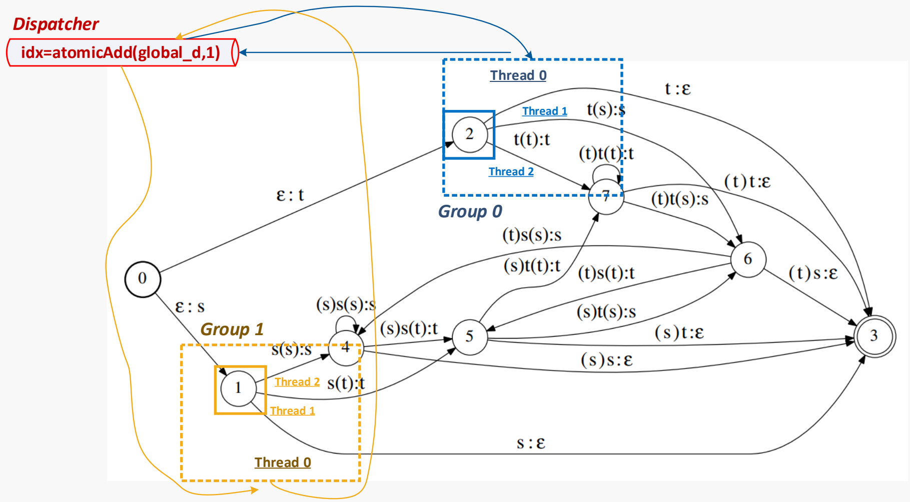
Group 0 with 32 threads
Token ?

Group 1 with 32 threads
Token ?



Load balancing

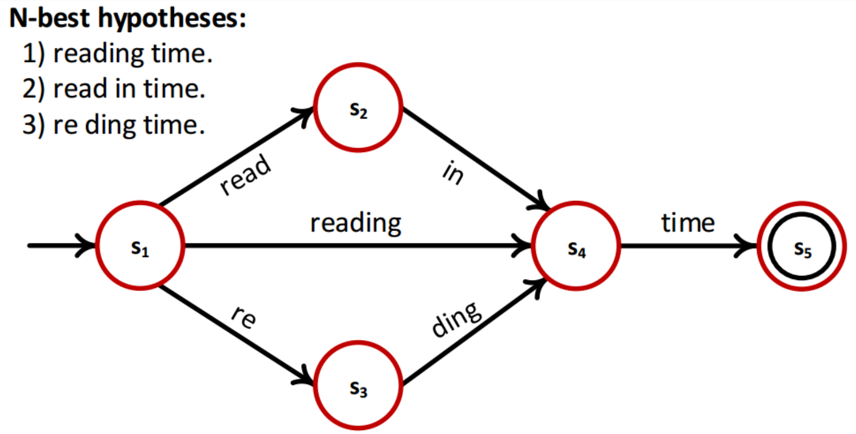
- Dynamic load balancing



Algorithm 2 Grid-level Token Passing (N=32; Inputs: the current active token vector)

procedure DYNAMIC LOAD BALANCING (toks)
    group = cooperative_groups::tiled_partition⟨32⟩
    if group.thread_rank()==0 **then** ▷ rank 0 in each group
        i = *atomicAdd*(global_d,1) ▷ allocate new tokens
    i = group.*shfl*(i,0) ▷ rank 0 broadcasts i to whole group
    if i>=sizeof(toks) **then** return ▷ all tokens processed
    for arc in tok2arcs(toks[i]) **do** ▷ thread parallelism
        call *Recombine*(toks[i].cost+arc.cost, arc, toks[i])

3rd Problem: Lattice processing



- Linkedlist → vector
- Atomic operations
  - e.g. memory allocation

```
// implementation of v.push_back(val)
int idx = atomicAdd(cnt_d, 1); // idx=cnt_d++
mem_d[idx] = *val; // store data
```

- Parallel lattice pruning

Experiments

- Setup
  - Switchboard 300 hours corpus, Cross Entropy & LF-MMI acoustic models (AM)
  - 30k-vocabulary, several tri-gram language models (LM)
- Baseline
  - Kaldi 1-best decoder
  - Kaldi lattice decoder
- GPU Optimization[1]
  - fast memcpy; merge GPU kernels by adding grid sync.; etc. (rel. 20% speedup)

---

1. https://github.com/chenzhehuai/kaldi/tree/gpu-decoder

Experiments

- Performance

Table 1: *1-best and Lattice Performance (beam=14).*

system	*lat. den.*	WER	*+rescored*	OWER	NCE
CPU	30.3	15.5	14.3	11.2	0.322
GPU	30.2	15.5	14.3	11.2	0.328

  - The same 1-best & lattice quality

Experiments

- Speedup

Table 2: *Speedup of the Proposed Method (beam=14).*

system	1-best RTF	1-best Δ	+ lattice RTF	+ lattice Δ
CPU	**0.16**	**1.0X**	**0.27**	**1.0X**
+ 8-sequence (1 socket)	-	-	0.15	1.8X
GPU	0.016	10X	0.080	3.3X
+ atomic operation	0.015	11X	0.077	3.5X
+ dyn. load balancing	**0.011**	**15X**	0.075	3.6X
+ lattice prune	-	-	**0.028**	**9.7X**
+ 8-sequence (MPS)	0.0035	46X	0.0080	34X

10X speedup

Experiments

- Speedup

Table 2: *Speedup of the Proposed Method (beam=14).*

system	1-best		+ lattice	
	RTF	Δ	RTF	Δ
CPU	**0.16**	**1.0X**	**0.27**	**1.0X**
+ 8-sequence (1 socket)	-	-	0.15	1.8X
GPU	0.016	10X	0.080	3.3X
+ atomic operation	0.015	11X	0.077	3.5X
+ dyn. load balancing	**0.011**	**15X**	0.075	3.6X
+ lattice prune	-	-	**0.028**	**9.7X**
+ 8-sequence (MPS)	0.0035	46X	0.0080	34X

+ utterance-parallelism

Experiments

- Speedup

Table 2: *Speedup of the Proposed Method (beam=14).*

system	1-best RTF	1-best Δ	+ lattice RTF	+ lattice Δ
CPU	**0.16**	**1.0X**	**0.27**	**1.0X**
+ 8-sequence (1 socket)	-	-	0.15	1.8X
GPU	0.016	10X	0.080	3.3X
+ atomic operation	0.015	11X	0.077	3.5X
+ dyn. load balancing	**0.011**	**15X**	0.075	3.6X
+ lattice prune	-	-	**0.028**	**9.7X**
+ 8-sequence (MPS)	0.0035	46X	0.0080	34X

+ utterance-parallelism

+ utterance-parallelism

Experiments

- Speedup

Table 2: *Speedup of the Proposed Method (beam=14).*

system	1-best RTF	1-best Δ	+ lattice RTF	+ lattice Δ
CPU	**0.16**	**1.0X**	**0.27**	**1.0X**
+ 8-sequence (1 socket)	-	-	0.15	1.8X
GPU	0.016	10X	0.080	3.3X
+ atomic operation	0.015	11X	0.077	3.5X
+ dyn. load balancing	**0.011**	**15X**	0.075	3.6X
+ lattice prune	-	-	**0.028**	**9.7X**
+ 8-sequence (MPS)	0.0035	46X	0.0080	34X

Improvement on naïve GPU decoder

Experiments

- Speedup

Table 2: *Speedup of the Proposed Method (beam=14).*

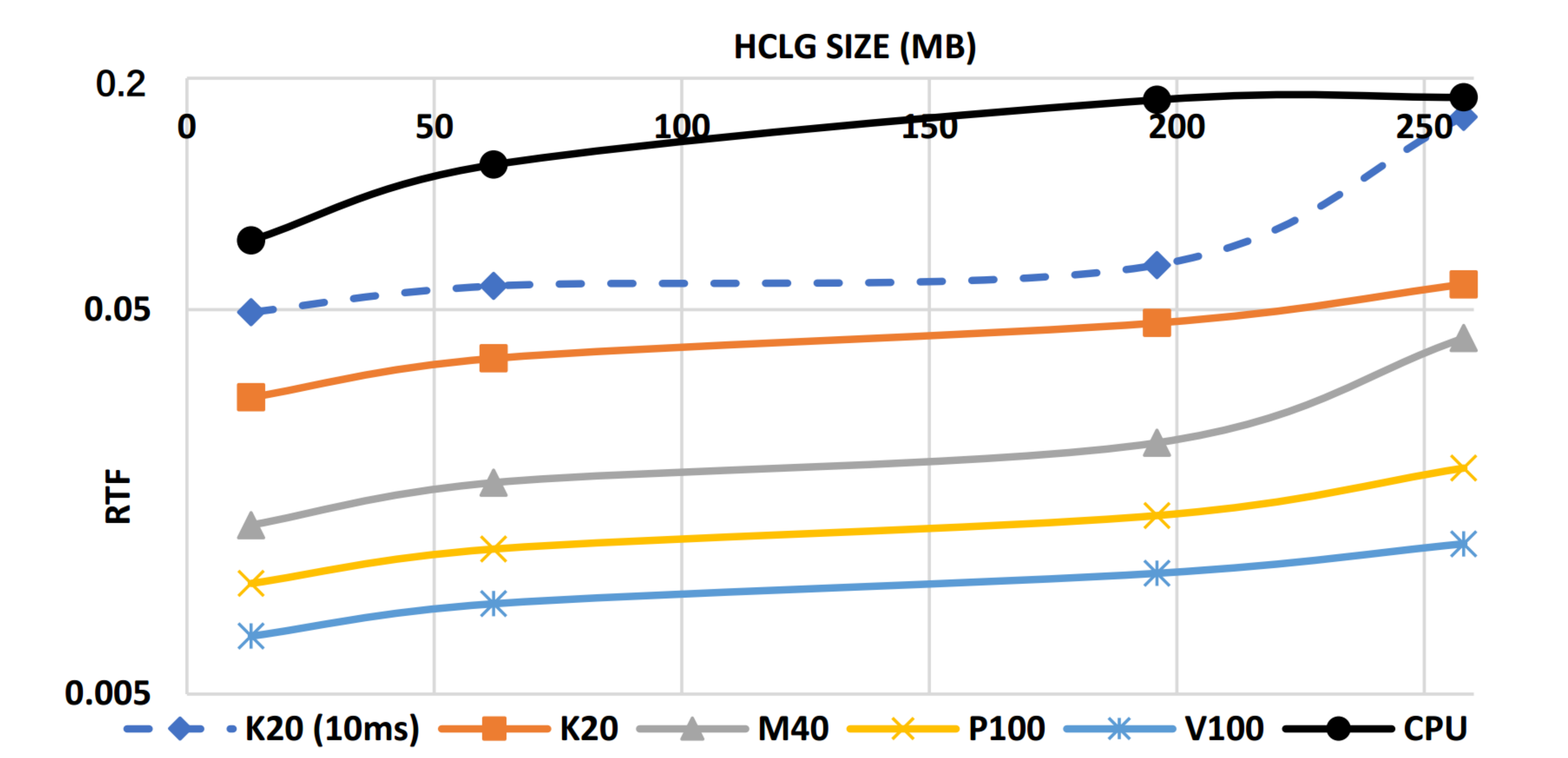
system	1-best RTF	1-best Δ	+ lattice RTF	+ lattice Δ
CPU	**0.16**	**1.0X**	**0.27**	**1.0X**
+ 8-sequence (1 socket)	-	-	0.15	1.8X
GPU	0.016	10X	0.080	3.3X
+ atomic operation	0.015	11X	0.077	3.5X
+ dyn. load balancing	**0.011**	**15X**	0.075	3.6X
+ lattice prune	-	-	**0.028**	**9.7X**
+ 8-sequence (MPS)	0.0035	46X	0.0080	34X

Our new number[1] is over 50X

1. https://github.com/chenzhehuai/kaldi/tree/gpu-decoder

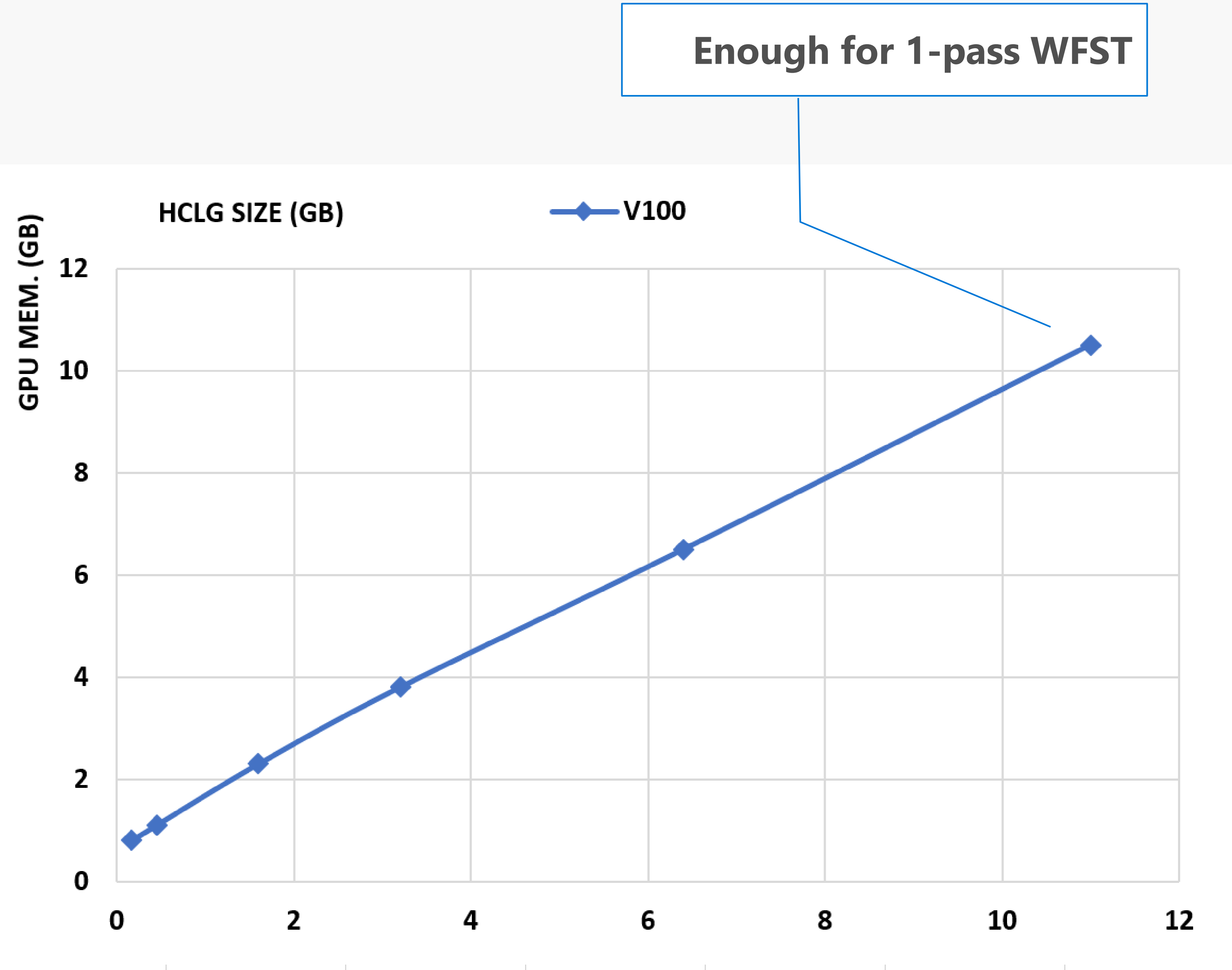
Experiments

- Compatibility
  - **GPU arch.**
  - **WFST size**
  - **Acoustic model**



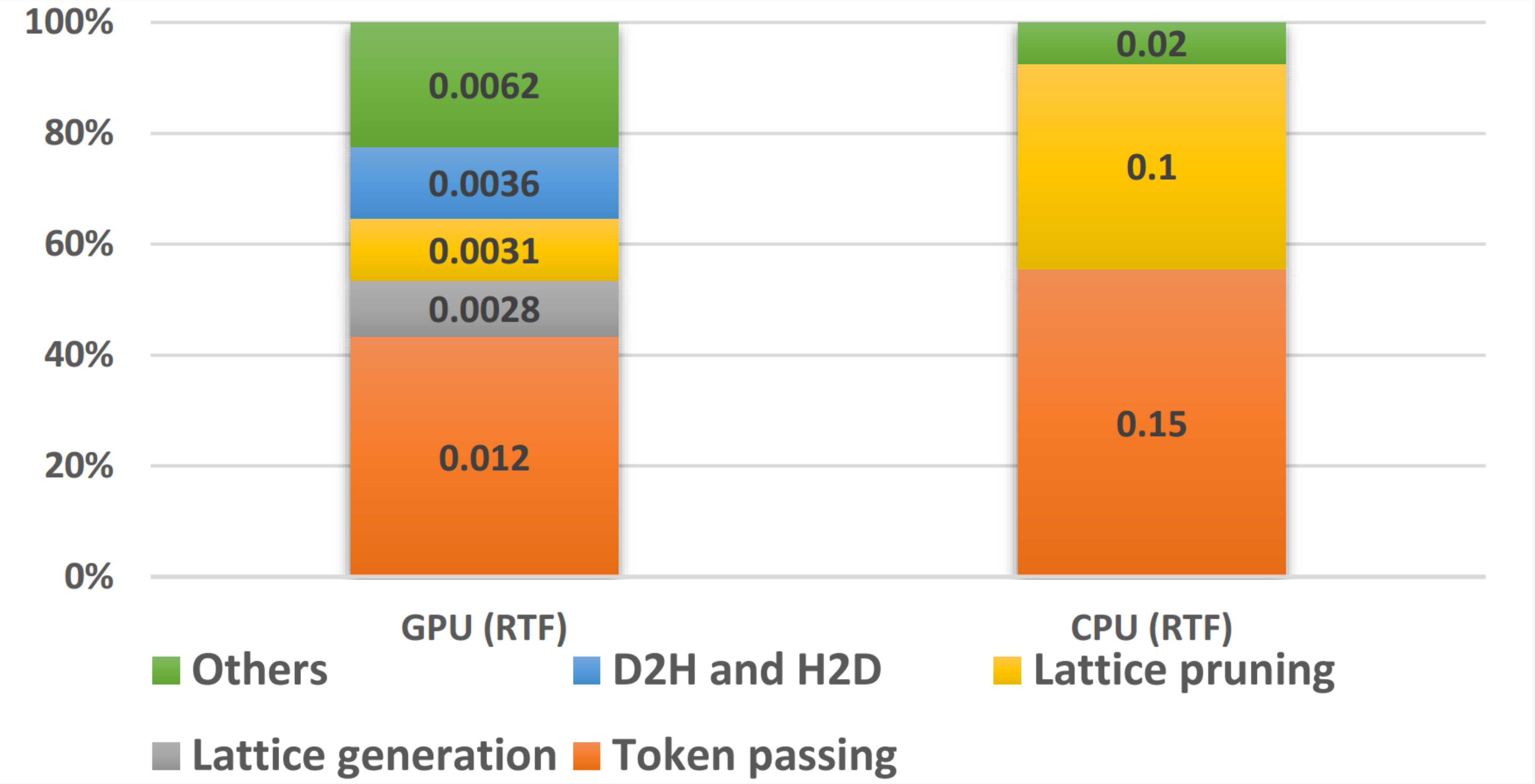
Experiments

- GPU Memory v.s. WFST size



Experiments

- Profiling



Conclusion & Future works

- Propose:
  - parallel Viterbi decoding & lattice processing
- Implementation:
  - Open-source & compatible with Kaldi recipes: https://github.com/chenzhehuai/kaldi/tree/gpu-decoder
- Future works:
  - More researches in GPU decoding
  - WFST algorithms, e.g. compose, determinize and minimize
  - Tight integration with acoustic inference (in GPUs)

Backup materials

Lattice processing

- Lattice pruning
  - The original CPU version: iteratively updates extra costs of nodes and arcs until they stop changing
  - Proposed:
    - updating in parallel
    - Linkedlist → vector
    - Atomic operations

Algorithm 3 Grid-level Lattice Processing (processing frame, lattice token vector and lattice arc vector are taken as inputs)

```
procedure PRUNE LATTICE FOR FRAME (f, toks, arcs)
    for tok in toks(f-1) do                    ▷ extraCost initialization
        tok.extraCost = FLT_MAX
    while modified == 1 do
        modified = 0
        for arc in arcs(f) do                  ▷ thread parallelism
            cost = ArcExtraCost(arc)
                ▷ returns the cost difference between the best
path including the arc, and the best overall path.
            if cost < latticeBeam then
                atomicMin(tok.extraCost,cost)
                atomicAdd(modified,1)
```

Speed and memory optimization

- share WFST between utterances in a GPU
- reduce context switching overheads: multi-process service (MPS)
- Lazy malloc to reduce memory: CudaMallocManaged
- grid sync implementation

```
__syncthreads();
if (threadIdx.x == 0) {
    int nb = 1;
    if (blockIdx.x == 0) {
        nb = 0x80000000 - (gridDim.x-1);
    }
    int old_epoch = *fast_epoch;
    __threadfence();
    atomicAdd((int*)fast_epoch, nb);
    int cnt=0;
    while (((*fast_epoch) ^ old_epoch) >= 0) ;
}
__syncthreads();
```

Speed and memory optimization

- reduce grid sync using multiple copy of variable
  - Do not need to make threads wait for modified1 = false

```
do {
    __grid_sync_nv_internal(params.barrier);

    swap(modified0,modified1);
    *modified1 = false;

    //__grid_sync_nv_internal(params.barrier);

    processNonEmittingTokens_function<32,2>(params,cutoff,size,modified0);
    __grid_sync_nv_internal(params.barrier);
} while ((*modified0)==true);
```

Speed and memory optimization

- reduce grid sync using multiple copy of variable
- faster atomicAdd by launch-and-go

```
//idx=atomicAdd((int*)fast_epoch, nb);
atomicAdd((int*)fast_epoch, nb);
```

- sharing atomicAdd
- fast copy

```
asm("st.global.v2.u64 [%0], {%1,%2};" :: "l"(a), "l"(src.x), "l"(src.y));
```