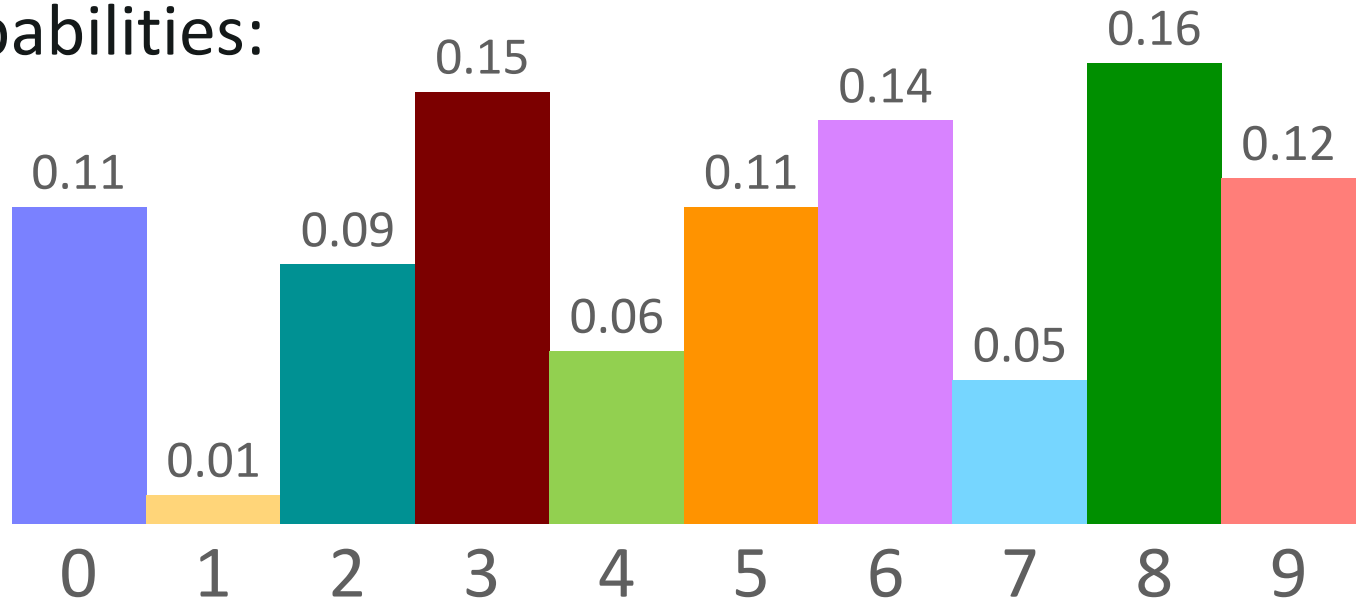
Using Butterfly-Patterned Partial Sums to Draw from Discrete Distributions

Guy L. Steele Jr.
Software Architect
Oracle Labs

ACM PPoPP Conference
February 8, 2017

The Task: Randomly Sample a Discrete Distribution

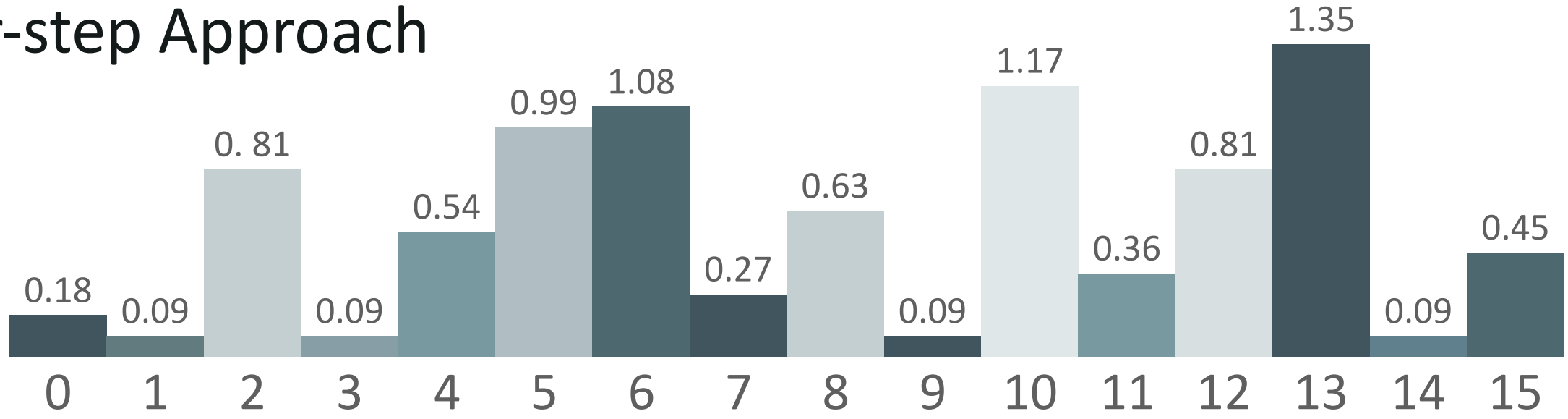
- Given a probability distribution described by a histogram or table of relative probabilities:



  randomly choose one of N integers (0 through N−1) accordingly

- This example table happens to be normalized (the sum of all entries is 1), but in general the table might not be normalized

Four-step Approach



- Start with an array **a** of the (possibly unnormalized) histogram entries:

  a =

0.18	0.09	0. 81	0.09	0.54	0.99	1.08	0.27	0.63	0.09	1.17	0.36	0.81	1.35	0.09	0.45

- Compute a new array **p** of the *running totals*:

  p =

0.18	0.27	1.08	1.17	1.71	2.70	3.78	4.05	4.68	4.77	5.94	6.30	7.11	8.46	8.55	9.00

- Choose random number **u** in [0.0,1.0) and multiply by final total **`p[N-1]`**
- *Search* **p** to find the smallest index **`j`** such that **`p[j] > u*p[N-1]`**

We Could Do a Linear Search

- It's easy:

```
u = value chosen uniformly randomly from [0.0, 1.0);
z = u * p[n-1];
j = 0;
while ((j < N-1) && (z >= p[k])) ++j;
```

this test is "mathematically unnecessary"
but defends against floating-point rounding errors
(alternatively, compute **z** in "round toward –∞" mode)

- But on a SIMD machine such as a GPU, it almost surely takes almost **N** steps (very likely that at least 1 of 32 warp threads will choose **u** close to 1)

But a Binary Search Takes Only O(log N) Time

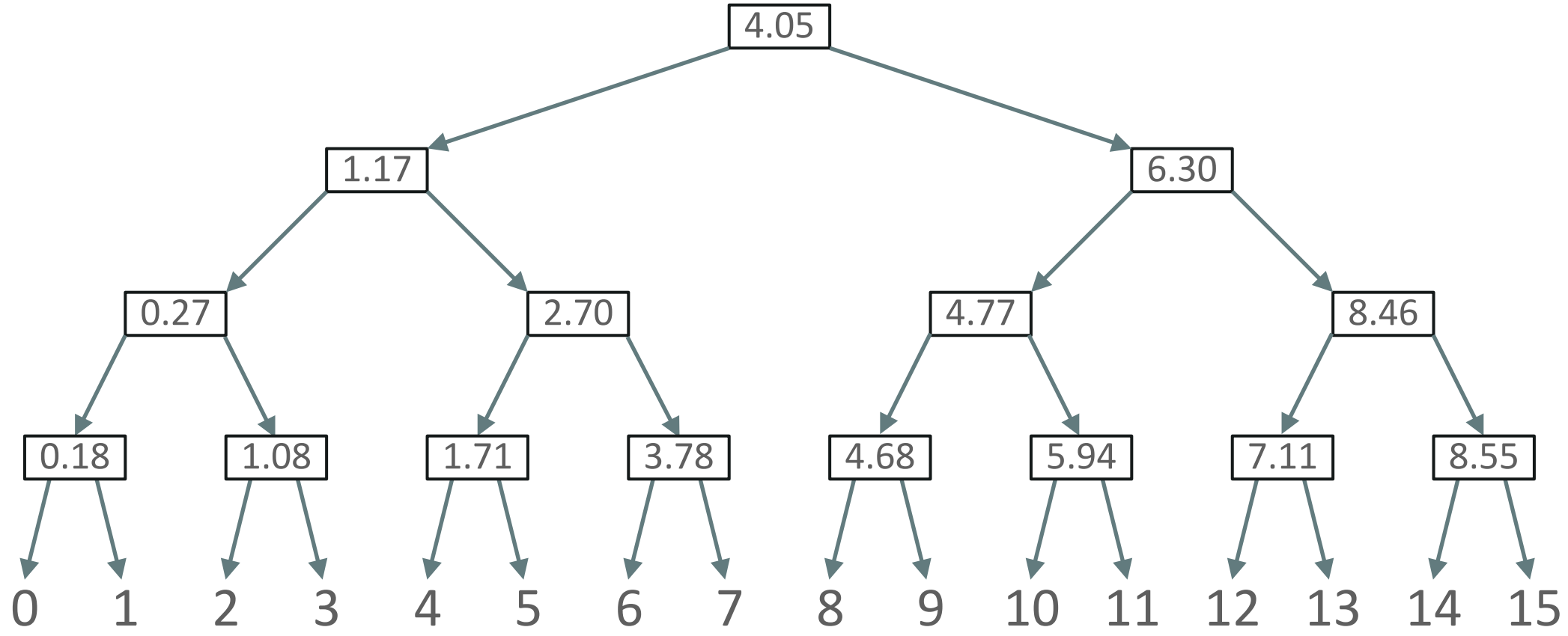
- Almost as easy:

```
u = value chosen uniformly randomly from [0.0, 1.0);
z = u * p[n-1];
j = 0;
k = N-1;
while (j < k) {
   m = (j + k) >> 1;     /* midpoint index between j and k */
   mid = p[m];           /* midpoint value */
   if (z < mid) k = m;
   else j = m+1;
}
```

A Binary Search Effectively Walks down a Tree

p =

0.18	0.27	1.08	1.17	1.71	2.70	3.78	4.05	4.68	4.77	5.94	6.30	7.11	8.46	8.55	9.00

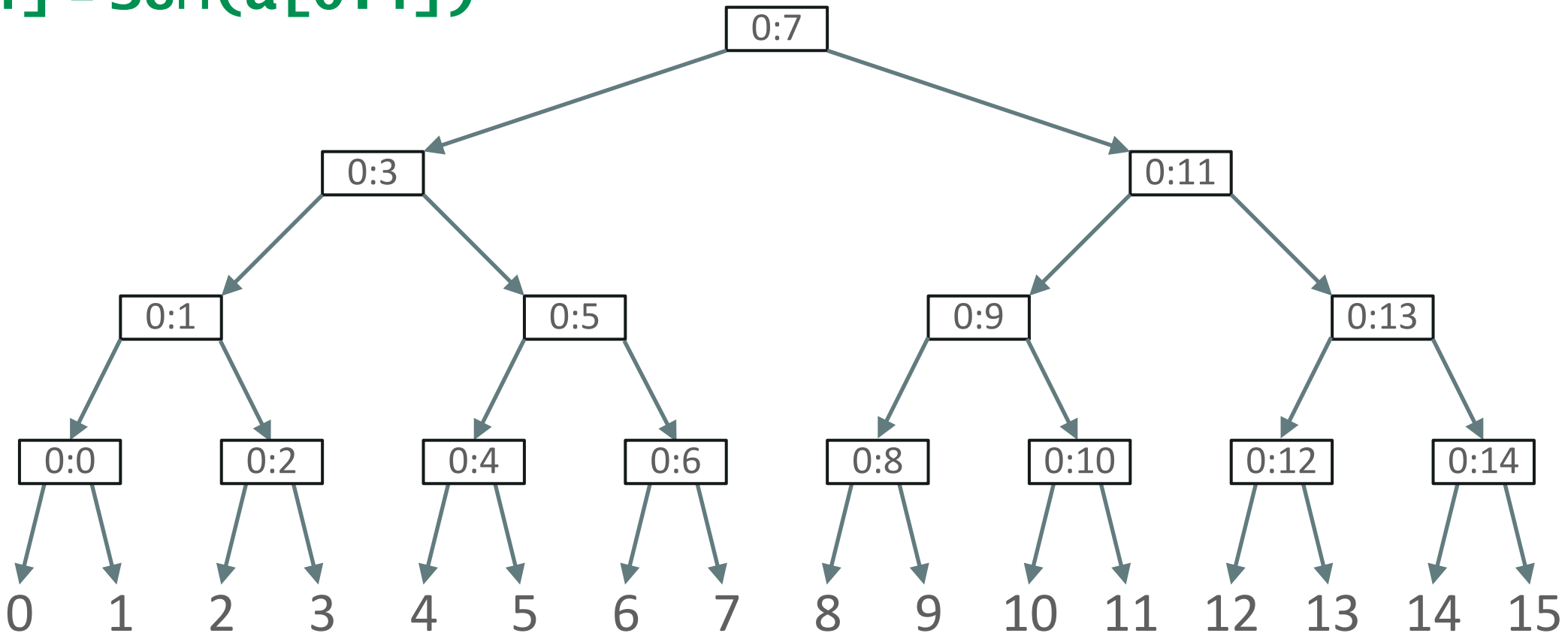


Abstraction of the Binary Tree, Showing Indexes

p =

0:0	0:1	0:2	0:3	0:4	0:5	0:6	0:7	0:8	0:9	0:10	0:11	0:12	0:13	0:14	0:15

```
p[i] = SUM(a[0:i])
```



Actually, We Want to Do Millions of These at Once

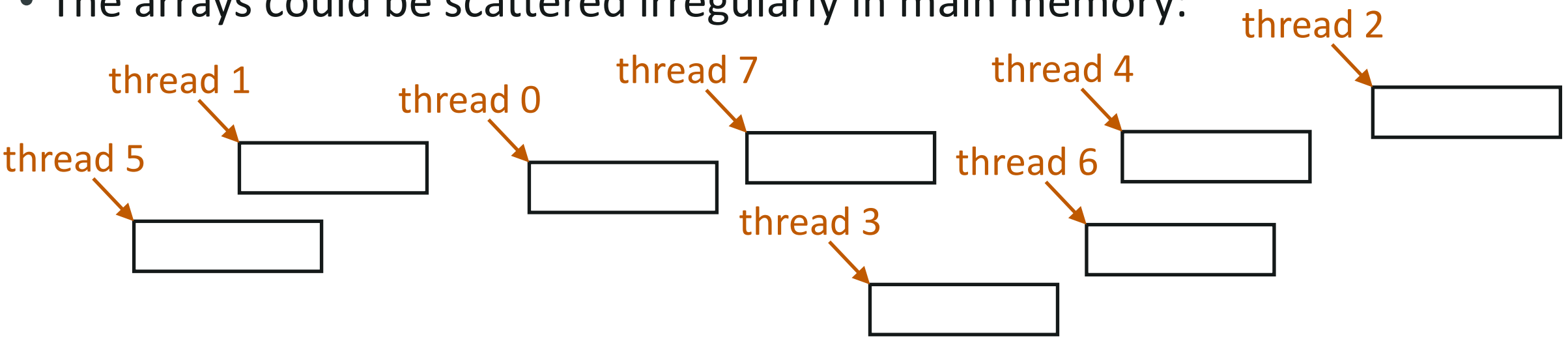
. . . with a different probability distribution for each one.

A GPU is a great way to do that!

BUT . . .

The Familiar “Memory Transposition” Problem

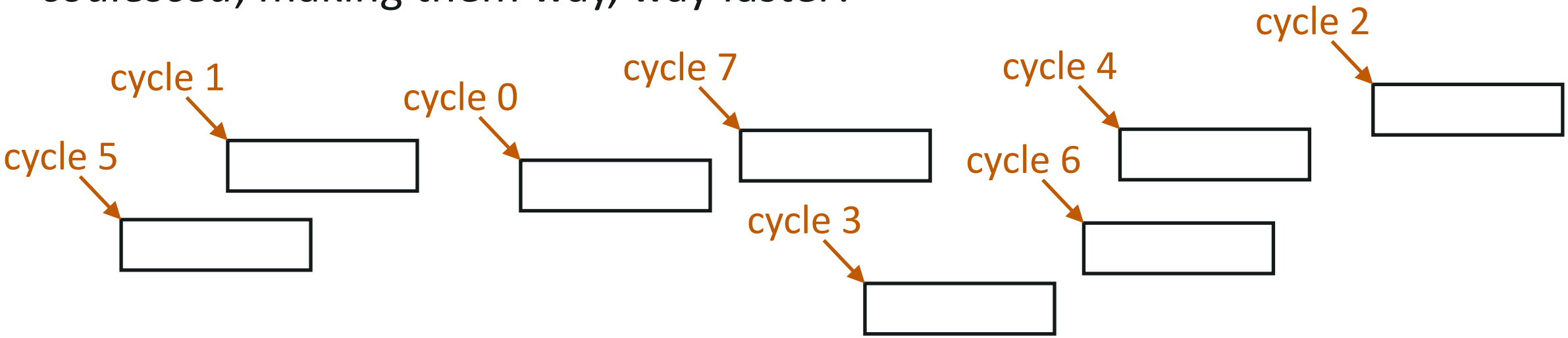
- Suppose every lane of a warp has a pointer to its own array **`a`** in memory
  - For now, assume the length **`N`** of each array **`a`** equals the number of lanes in a warp
- The arrays could be scattered irregularly in main memory:



- If on cycle **`j`** ($0 \le$ **`j`** $<$ **`N`**), lane **`k`** of the warp fetches its own **`a[j]`**, then you can get really bad memory performance

Transposed Memory Access

- Repeated from previous slide:
  If on cycle **`j`** (0 ≤ **`j`** < **N**), lane **`k`** of the warp fetches its own **`a[j]`**,
  then you can get really bad memory performance
- But if on cycle **`j`**, lane **`k`** fetches the **`a[k]`** for thread **`j`**, then together they are fetching **N** *consecutive* memory words; the memory accesses can be *coalesced*, making them way, way faster!



Now we have all the data in lane registers . . .

lanes	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}	A_{13}	A_{14}	A_{15}
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10}	B_{11}	B_{12}	B_{13}	B_{14}	B_{15}
	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}	C_{11}	C_{12}	C_{13}	C_{14}	C_{15}
	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}	D_{11}	D_{12}	D_{13}	D_{14}	D_{15}
	E_0	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9	E_{10}	E_{11}	E_{12}	E_{13}	E_{14}	E_{15}
	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
	G_0	G_1	G_2	G_3	G_4	G_5	G_6	G_7	G_8	G_9	G_{10}	G_{11}	G_{12}	G_{13}	G_{14}	G_{15}
	H_0	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8	H_9	H_{10}	H_{11}	H_{12}	H_{13}	H_{14}	H_{15}
	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}	I_{11}	I_{12}	I_{13}	I_{14}	I_{15}
	J_0	J_1	J_2	J_3	J_4	J_5	J_6	J_7	J_8	J_9	J_{10}	J_{11}	J_{12}	J_{13}	J_{14}	J_{15}
	K_0	K_1	K_2	K_3	K_4	K_5	K_6	K_7	K_8	K_9	K_{10}	K_{11}	K_{12}	K_{13}	K_{14}	K_{15}
	L_0	L_1	L_2	L_3	L_4	L_5	L_6	L_7	L_8	L_9	L_{10}	L_{11}	L_{12}	L_{13}	L_{14}	L_{15}
	M_0	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}	M_{11}	M_{12}	M_{13}	M_{14}	M_{15}
	N_0	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_9	N_{10}	N_{11}	N_{12}	N_{13}	N_{14}	N_{15}
	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7	O_8	O_9	O_{10}	O_{11}	O_{12}	O_{13}	O_{14}	O_{15}
	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}

Each column represents the registers used to hold the array **a** for one lane.

but what we really wanted was this:

lanes	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
	A_0	B_0	C_0	D_0	E_0	F_0	G_0	H_0	I_0	J_0	K_0	L_0	M_0	N_0	O_0	P_0
	A_1	B_1	C_1	D_1	E_1	F_1	G_1	H_1	I_1	J_1	K_1	L_1	M_1	N_1	O_1	P_1
	A_2	B_2	C_2	D_2	E_2	F_2	G_2	H_2	I_2	J_2	K_2	L_2	M_2	N_2	O_2	P_2
	A_3	B_3	C_3	D_3	E_3	F_3	G_3	H_3	I_3	J_3	K_3	L_3	M_3	N_3	O_3	P_3
	A_4	B_4	C_4	D_4	E_4	F_4	G_4	H_4	I_4	J_4	K_4	L_4	M_4	N_4	O_4	P_4
	A_5	B_5	C_5	D_5	E_5	F_5	G_5	H_5	I_5	J_5	K_5	L_5	M_5	N_5	O_5	P_5
	A_6	B_6	C_6	D_6	E_6	F_6	G_6	H_6	I_6	J_6	K_6	L_6	M_6	N_6	O_6	P_6
	A_7	B_7	C_7	D_7	E_7	F_7	G_7	H_7	I_7	J_7	K_7	L_7	M_7	N_7	O_7	P_7
	A_8	B_8	C_8	D_8	E_8	F_8	G_8	H_8	I_8	J_8	K_8	L_8	M_8	N_8	O_8	P_8
	A_9	B_9	C_9	D_9	E_9	F_9	G_9	H_9	I_9	J_9	K_9	L_9	M_9	N_9	O_9	P_9
	A_{10}	B_{10}	C_{10}	D_{10}	E_{10}	F_{10}	G_{10}	H_{10}	I_{10}	J_{10}	K_{10}	L_{10}	M_{10}	N_{10}	O_{10}	P_{10}
	A_{11}	B_{11}	C_{11}	D_{11}	E_{11}	F_{11}	G_{11}	H_{11}	I_{11}	J_{11}	K_{11}	L_{11}	M_{11}	N_{11}	O_{11}	P_{11}
	A_{12}	B_{12}	C_{12}	D_{12}	E_{12}	F_{12}	G_{12}	H_{12}	I_{12}	J_{12}	K_{12}	L_{12}	M_{12}	N_{12}	O_{12}	P_{12}
	A_{13}	B_{13}	C_{13}	D_{13}	E_{13}	F_{13}	G_{13}	H_{13}	I_{13}	J_{13}	K_{13}	L_{13}	M_{13}	N_{13}	O_{13}	P_{13}
	A_{14}	B_{14}	C_{14}	D_{14}	E_{14}	F_{14}	G_{14}	H_{14}	I_{14}	J_{14}	K_{14}	L_{14}	M_{14}	N_{14}	O_{14}	P_{14}
	A_{15}	B_{15}	C_{15}	D_{15}	E_{15}	F_{15}	G_{15}	H_{15}	I_{15}	J_{15}	K_{15}	L_{15}	M_{15}	N_{15}	O_{15}	P_{15}

Each column represents the registers used to hold the array **a** for one lane.

Transposing Registers in Time *O*(**N**) (1 of 2)

- Now every thread of the warp has **N** array values in its registers—but all of them except one belong to some other thread!
- What we would like to say now is:

```
int k = threadIdx.x & 0x1f;          /*  lane ID  */
for (int j = 0; j < N; j++) {
  int m = (j-k)&(N-1);               /*  that is, (j–k) mod N  */
  reg[m] = __shfl(reg[m], m);
}
```

which would get all the data where it belongs in (small) constant time

Transposing Registers in Time *O*(**N**) (2 of 2)

- Now every thread of the warp has **N** array values in its registers—but all of them except one belong to some other thread!
- What we would like to say is:

```
int k = threadIdx.x & 0x1f;        /*  lane ID  */
for (int j = 0; j < N; j++) {
  int m = (j-k)&(N-1);             /*  that is, (j–k) mod N  */
  reg[m] = __shfl(reg[m], m);
}
```

  which would get all the data where it belongs in (small) constant time

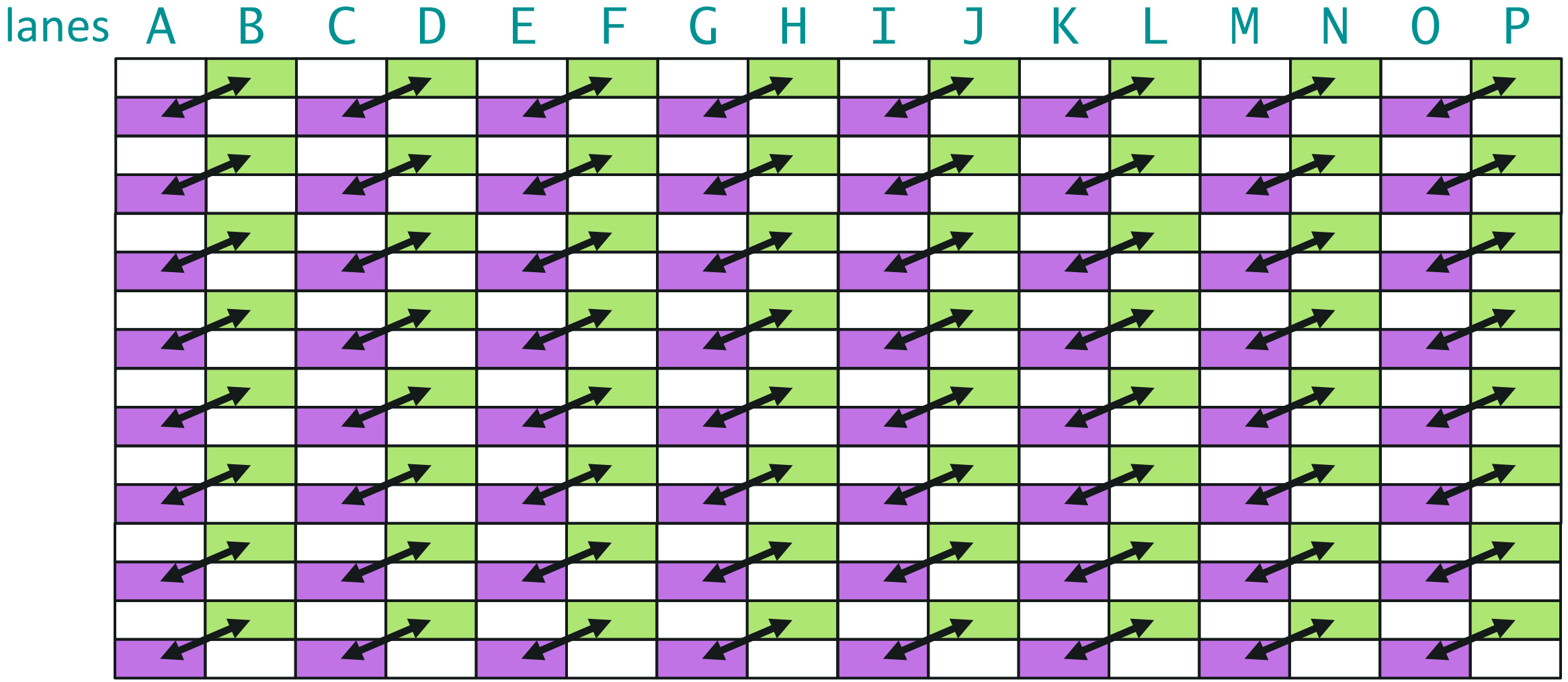
- Alas, typical GPU instruction sets don't support

lane-dependent register indexing

Transposing Registers in Time *O*(**N** log **N**)

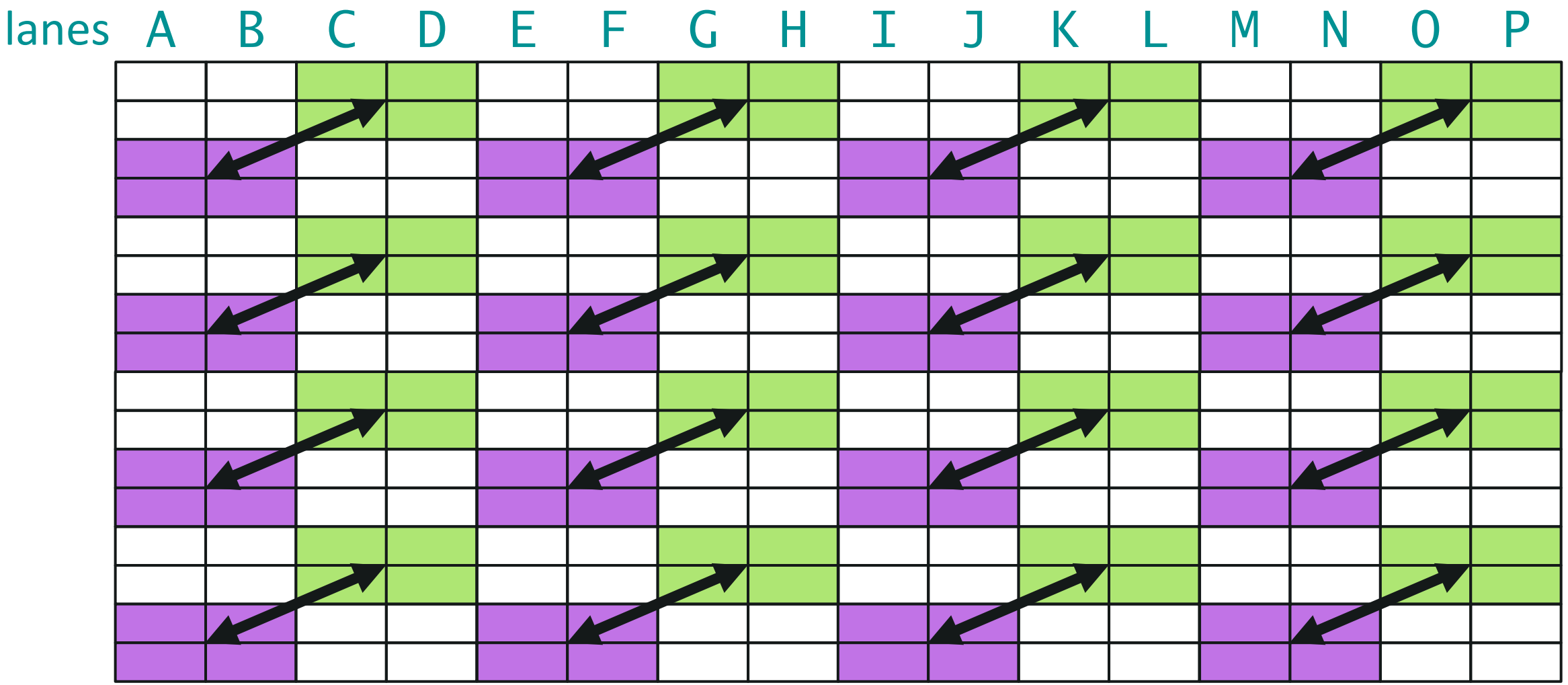
- There's a great trick that uses **`__shfl_xor`**, but it takes *O*(log **N**) passes: on pass **b** (0 ≤ **b** < log **N**), lane **k** exchanges data with lane **`(k^(1<<b))`** (this is the "hypercube exchange pattern")
  - You still need to "index" into the registers, but at each step it's just a two-way choice rather than an **N**-way choice, so you can use a conditional expression with **`? :`**
  - For **N** = 32, it requires 5*(32/2) = 80 shuffle operations

Transposing Registers: Step 1



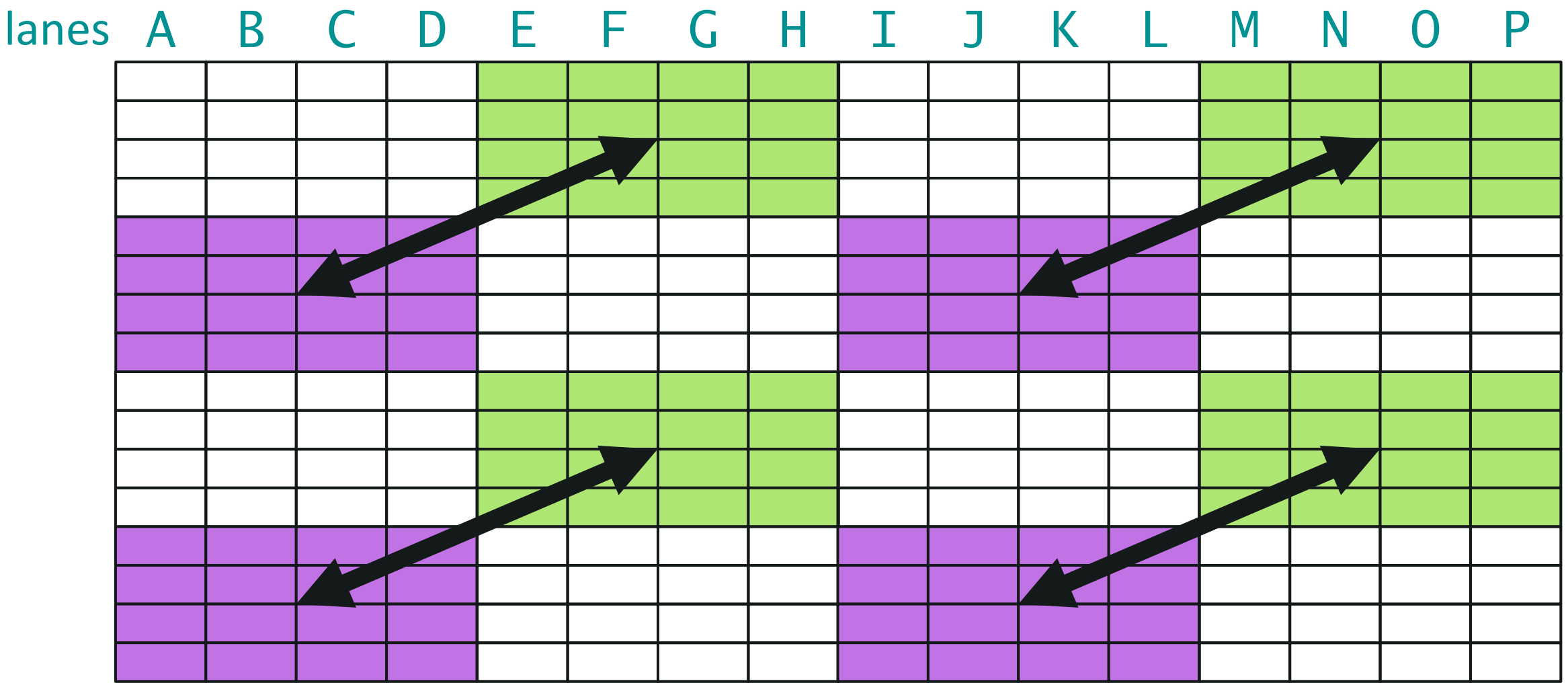
Each column represents the registers used to hold the array **a** for one lane.

Transposing Registers: Step 2



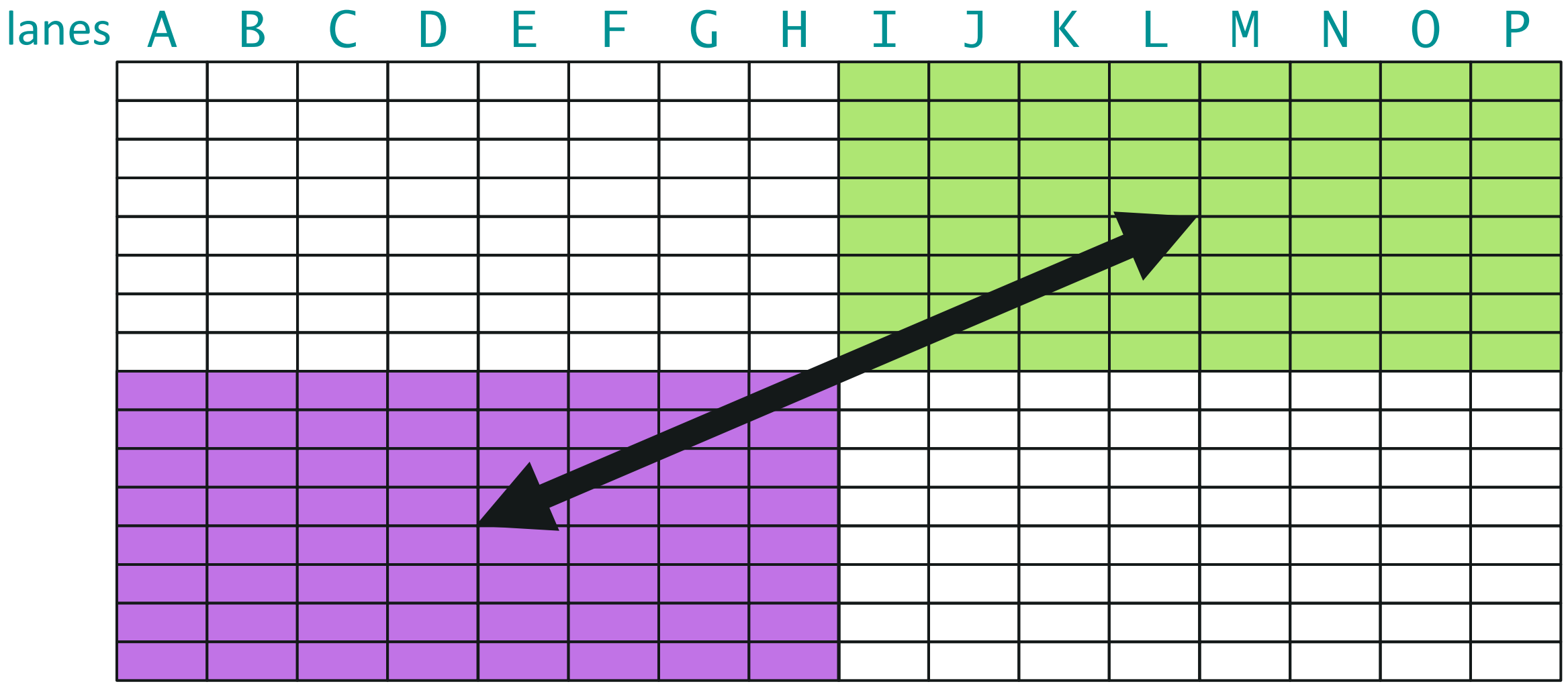
Each column represents the registers used to hold the array **a** for one lane.

Transposing Registers: Step 3 . . .



Each column represents the registers used to hold the array **a** for one lane.

Transposing Registers: Step (log **N**)



Each column represents the registers used to hold the array **a** for one lane.

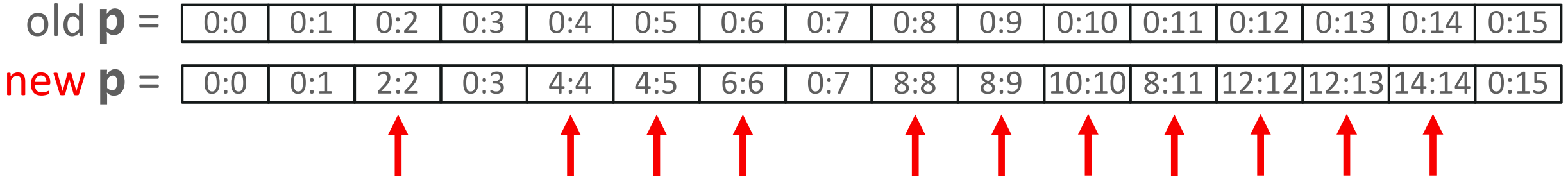
Transposing Registers Takes Too Much Time

- There's a great trick that uses `__shfl_xor`, but it takes *O*(log **N**) passes: on pass **b** (0 ≤ **b** < log **N**), lane **k** exchanges data with lane `(k^(1<<b))`
  - You still need to "index" into the registers, but at each step it's just a two-way choice rather than an **N**-way choice, so you can use a conditional expression with `? :`
  - For **N** = 32, it requires 5*(32/2) = 80 shuffle operations
- It turns out to be slightly better to just dump the registers into a shared-memory array and then reload them (using appropriately indexed accesses)
- But neither of these tricks was good enough (they were both too slow)
- Transposing registers became the critical performance bottleneck in the inner loop of our machine-learning application (topic modeling by using Gibbs sampling on a Latent Dirichlet Allocation model)

Idea: Simpler Partial Sums (1 of 4)

old **p** =

Idea: Simpler Partial Sums (2 of 4)



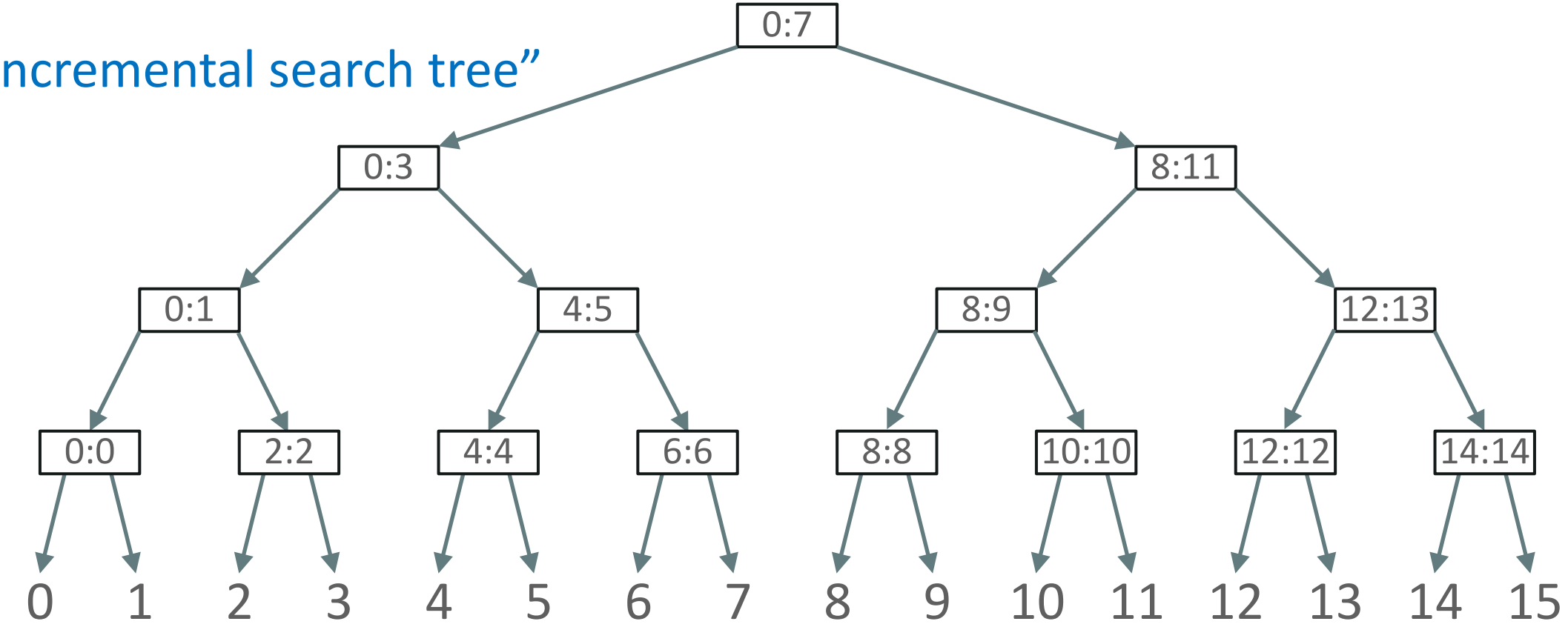
Idea: Simpler Partial Sums (3 of 4)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
old **p** =	0:0	0:1	0:2	0:3	0:4	0:5	0:6	0:7	0:8	0:9	0:10	0:11	0:12	0:13	0:14	0:15
new **p** =	0:0	0:1	2:2	0:3	4:4	4:5	6:6	0:7	8:8	8:9	10:10	8:11	12:12	12:13	14:14	0:15
	1	2	1	4	1	2	1	8	1	2	1	4	1	2	1	16

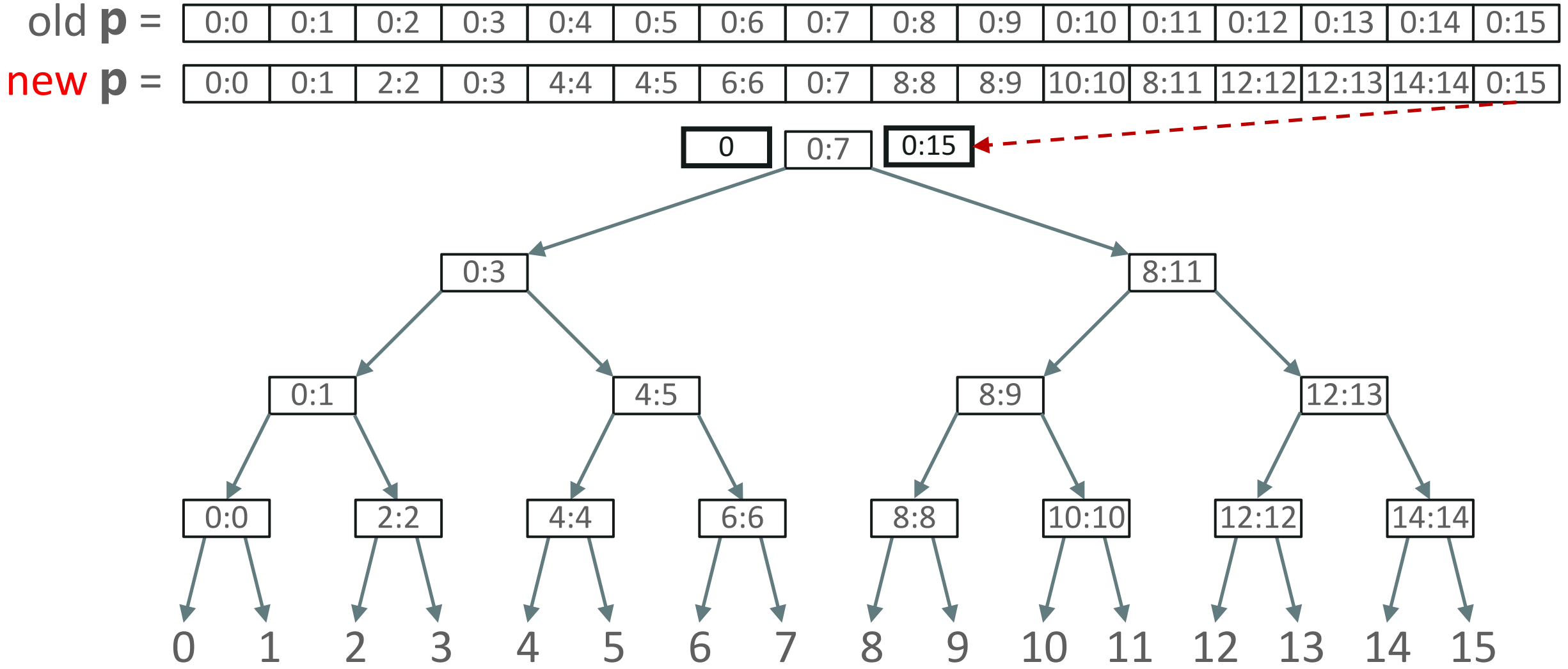
Idea: Simpler Partial Sums (4 of 4)

old **p** =	0:0	0:1	0:2	0:3	0:4	0:5	0:6	0:7	0:8	0:9	0:10	0:11	0:12	0:13	0:14	0:15
new **p** =	0:0	0:1	2:2	0:3	4:4	4:5	6:6	0:7	8:8	8:9	10:10	8:11	12:12	12:13	14:14	0:15

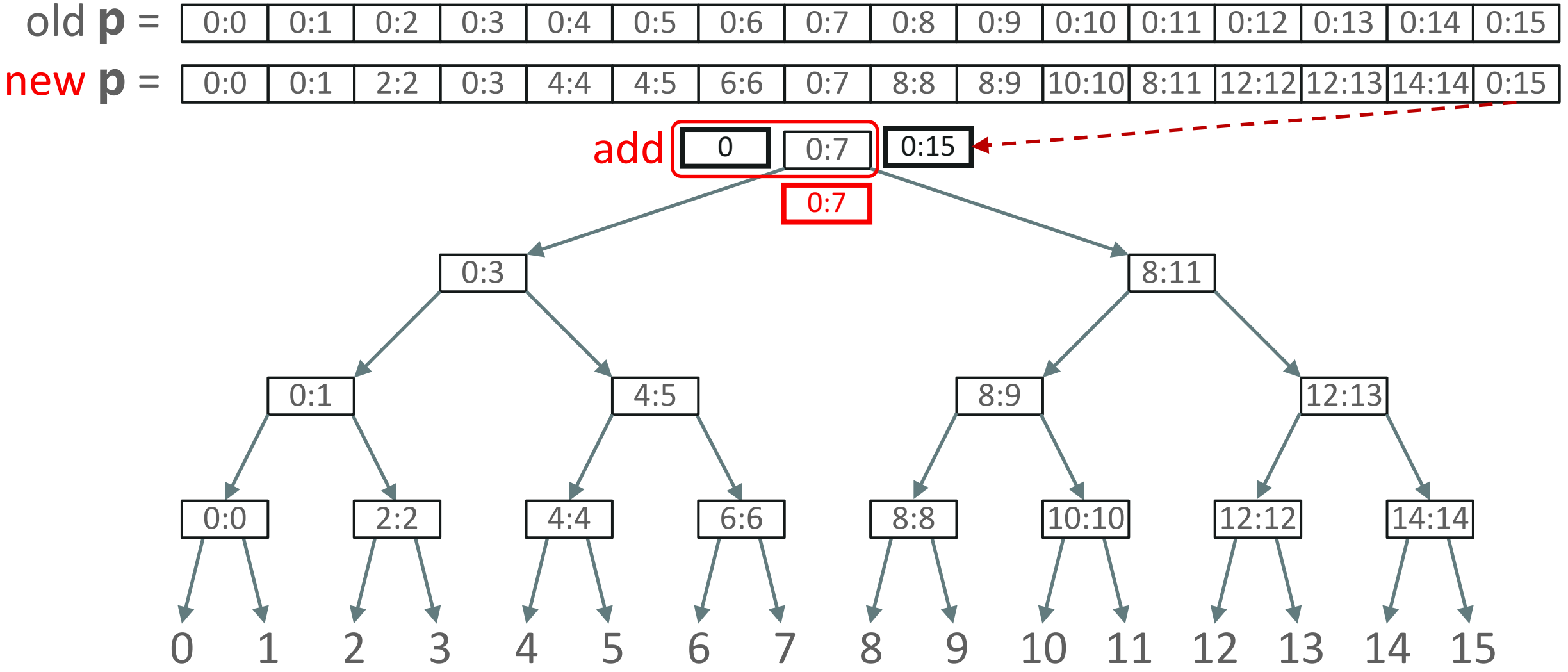
"Incremental search tree"



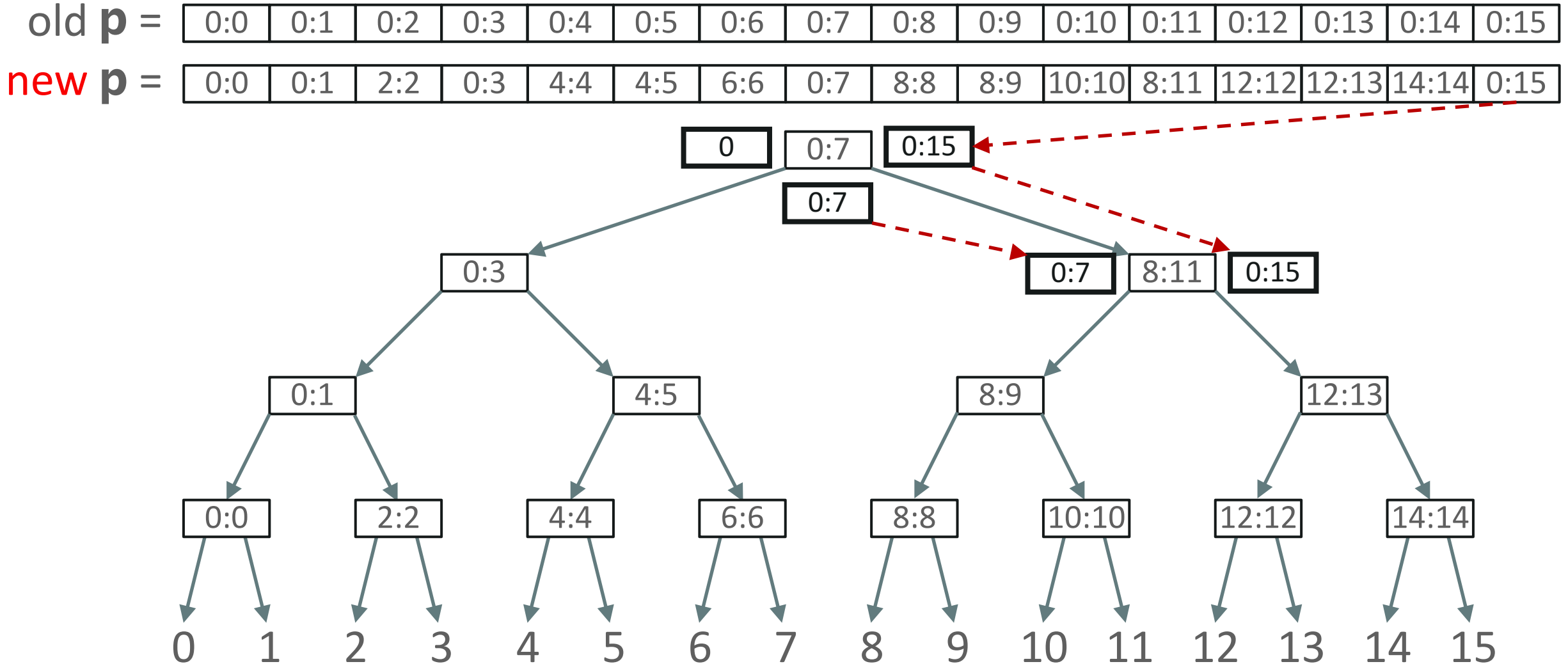
Searching Simpler Partial Sums (1 of 6)



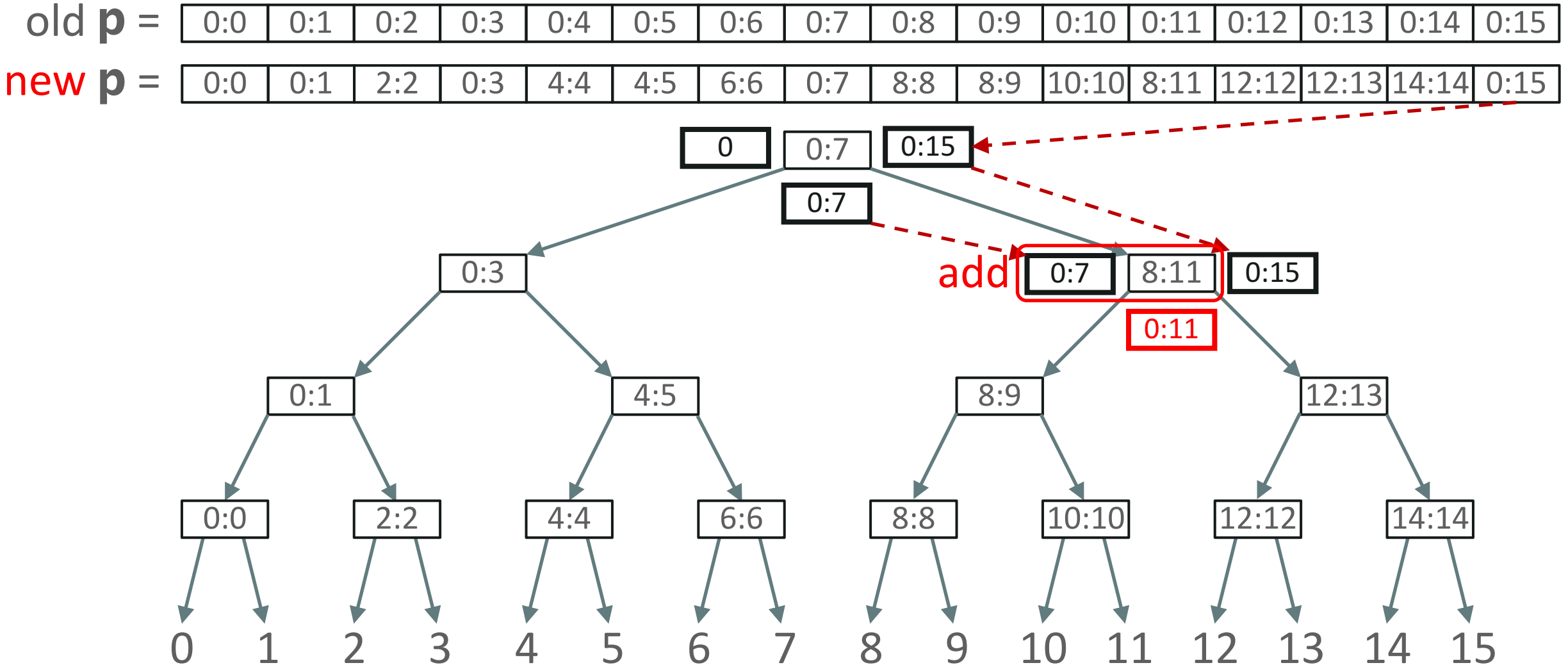
Searching Simpler Partial Sums (2 of 6)



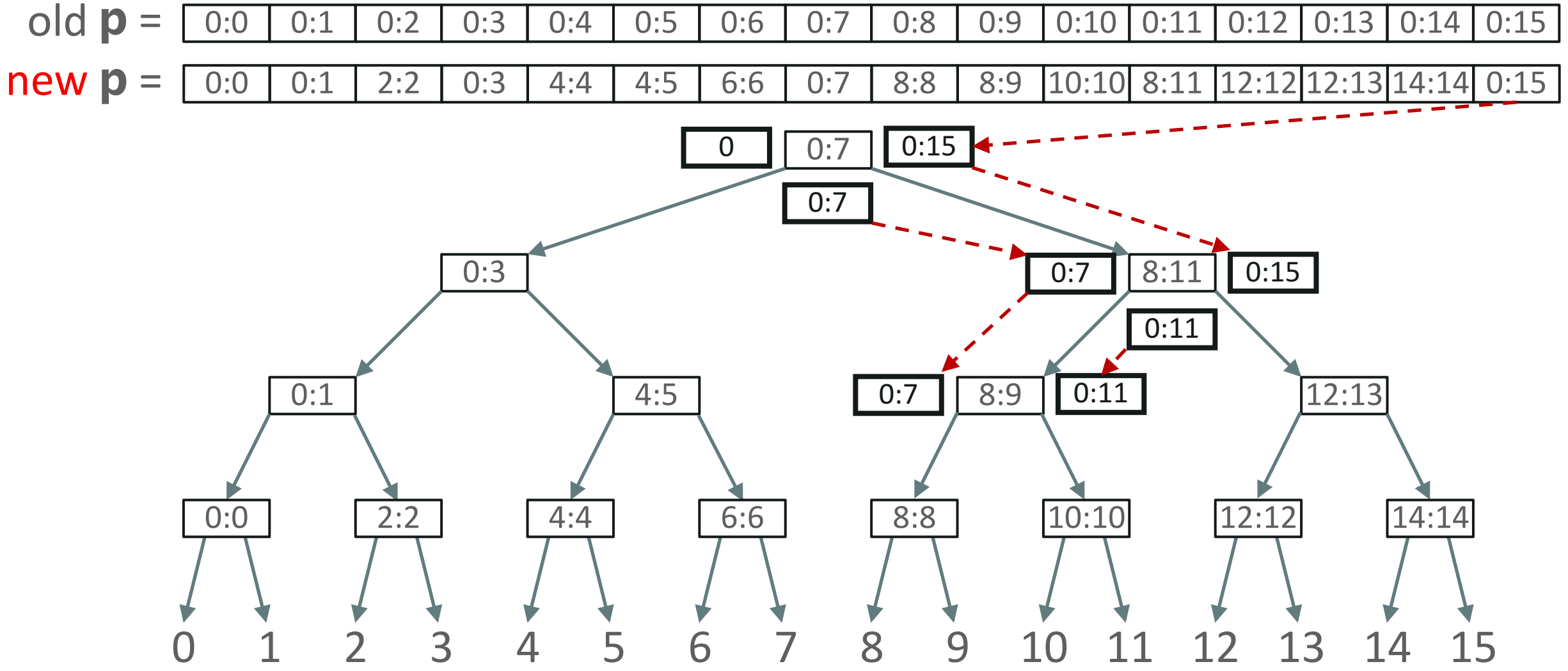
Searching Simpler Partial Sums (3 of 6)



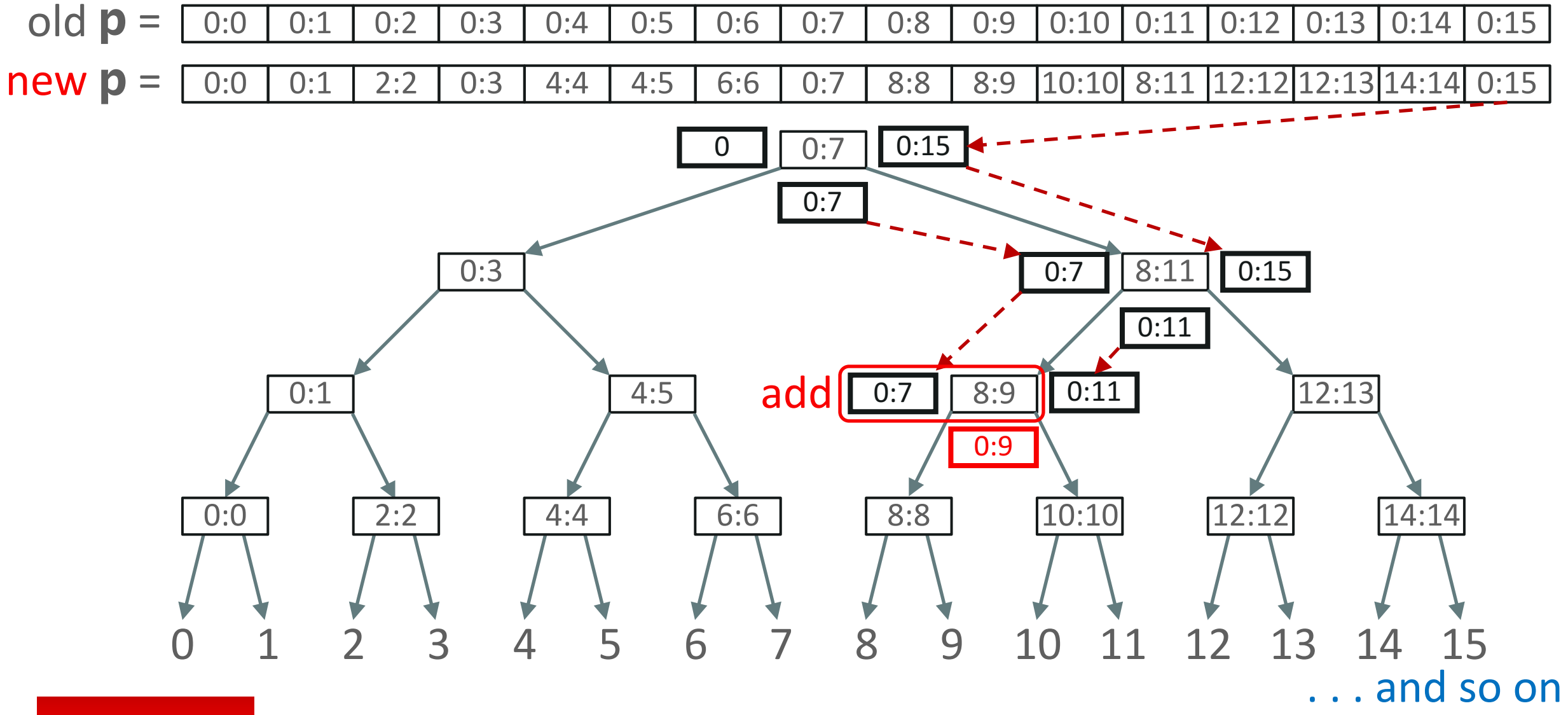
Searching Simpler Partial Sums (4 of 6)



Searching Simpler Partial Sums (5 of 6)



Searching Simpler Partial Sums (6 of 6)



The Binary Search Does a Little Extra Work

- Also easy:

```
u = value chosen uniformly randomly from [0.0, 1.0);
z = u * p[n-1];
j = 0;     lo = 0;
k = N-1;   hi = p[N-1];
while (j < k) {
    m = (j + k) >> 1;
    mid = lo + p[m];
    if (z < mid) { k = m;   hi = mid; }
    else         { j = m+1; lo = mid; }
}
```

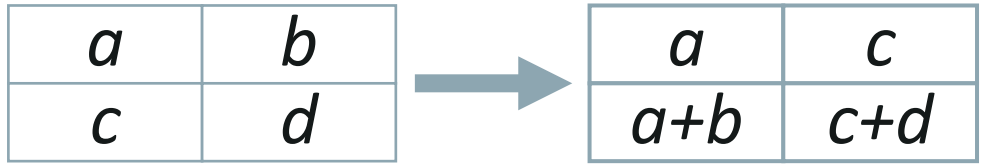
The Binary Search Does a Little Extra Work

- Also easy:

```
u = value chosen uniformly randomly from [0.0, 1.0);
z = u * p[n-1];
j = 0;     lo = 0;
k = N-1;   hi = p[N-1];
while (j < k) {
    m = (j + k) >> 1;
    mid = lo + p[m];
    if (z < mid) { k = m;     hi = mid; }
    else         { j = m+1;   lo = mid; }
}
```

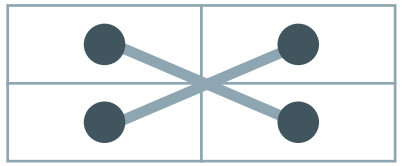
The Big Idea: Creating N Such Incremental Trees in One Pass

- Do "butterfly" (**__shfl_xor**) exchanges using this computation pattern:



That is, swap *b* and *c*, then add first row into second row

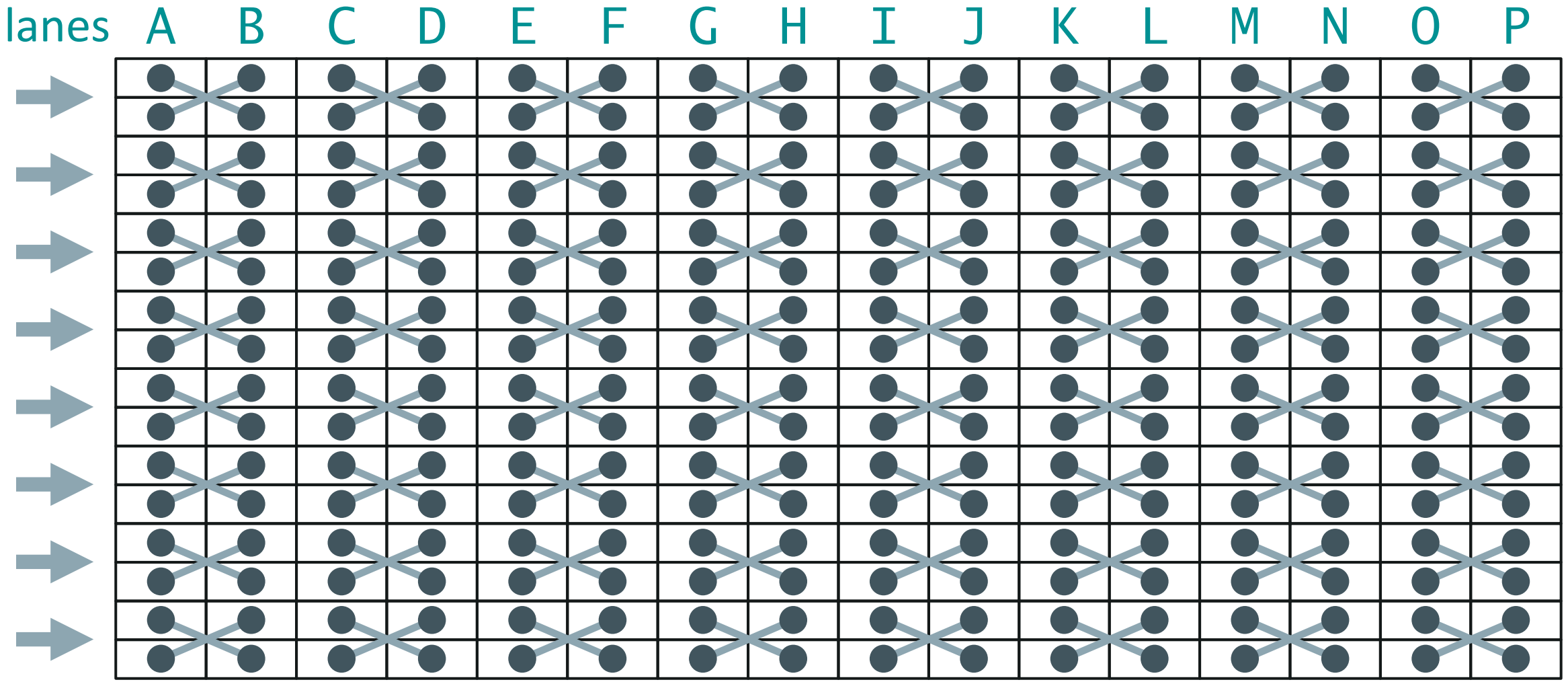
- We will symbolize applying this pattern to a group of four registers (two registers in each of two lanes) as simply:



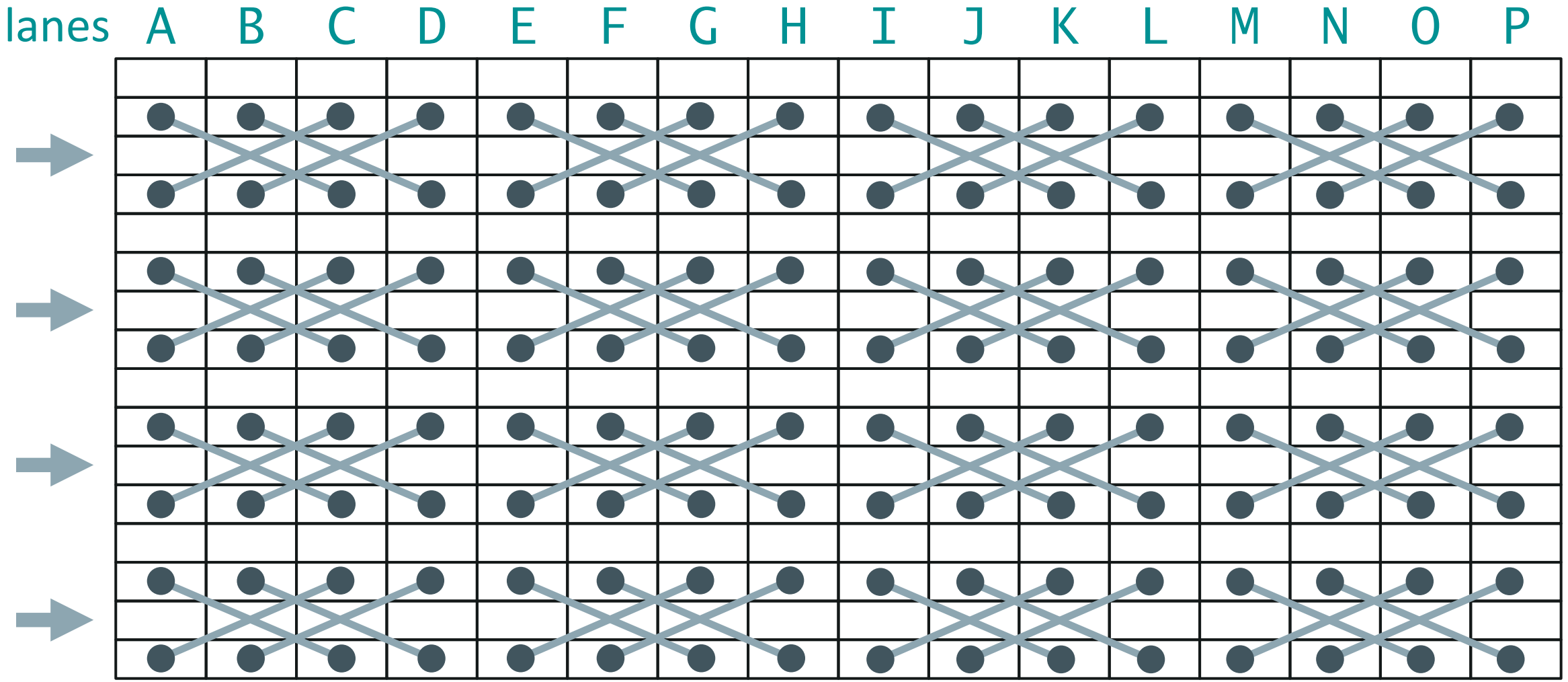
Here's the Transposed Memory Data

lanes	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}	A_{13}	A_{14}	A_{15}
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10}	B_{11}	B_{12}	B_{13}	B_{14}	B_{15}
	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}	C_{11}	C_{12}	C_{13}	C_{14}	C_{15}
	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}	D_{11}	D_{12}	D_{13}	D_{14}	D_{15}
	E_0	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9	E_{10}	E_{11}	E_{12}	E_{13}	E_{14}	E_{15}
	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
	G_0	G_1	G_2	G_3	G_4	G_5	G_6	G_7	G_8	G_9	G_{10}	G_{11}	G_{12}	G_{13}	G_{14}	G_{15}
	H_0	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8	H_9	H_{10}	H_{11}	H_{12}	H_{13}	H_{14}	H_{15}
	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}	I_{11}	I_{12}	I_{13}	I_{14}	I_{15}
	J_0	J_1	J_2	J_3	J_4	J_5	J_6	J_7	J_8	J_9	J_{10}	J_{11}	J_{12}	J_{13}	J_{14}	J_{15}
	K_0	K_1	K_2	K_3	K_4	K_5	K_6	K_7	K_8	K_9	K_{10}	K_{11}	K_{12}	K_{13}	K_{14}	K_{15}
	L_0	L_1	L_2	L_3	L_4	L_5	L_6	L_7	L_8	L_9	L_{10}	L_{11}	L_{12}	L_{13}	L_{14}	L_{15}
	M_0	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}	M_{11}	M_{12}	M_{13}	M_{14}	M_{15}
	N_0	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_9	N_{10}	N_{11}	N_{12}	N_{13}	N_{14}	N_{15}
	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7	O_8	O_9	O_{10}	O_{11}	O_{12}	O_{13}	O_{14}	O_{15}
	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}

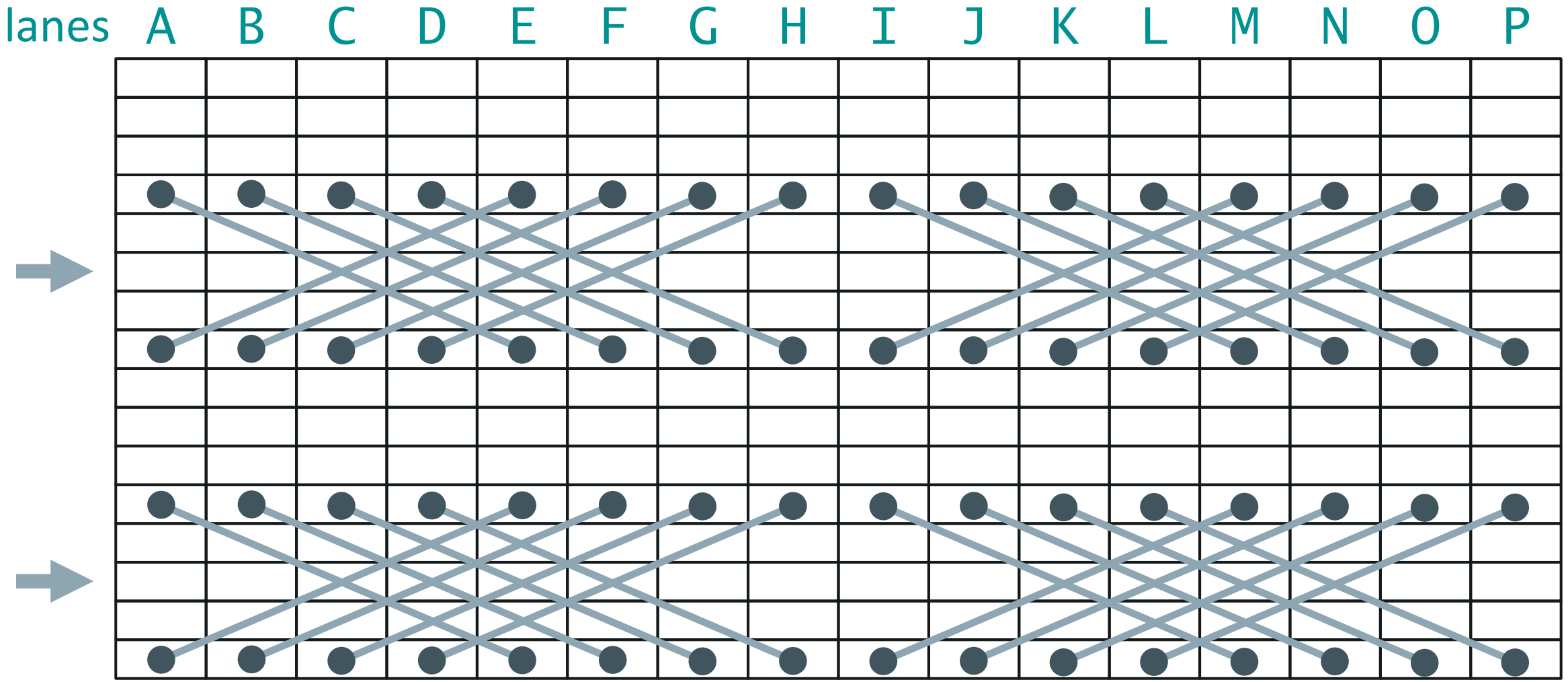
Now do eight **__shfl_xor** instructions . . .



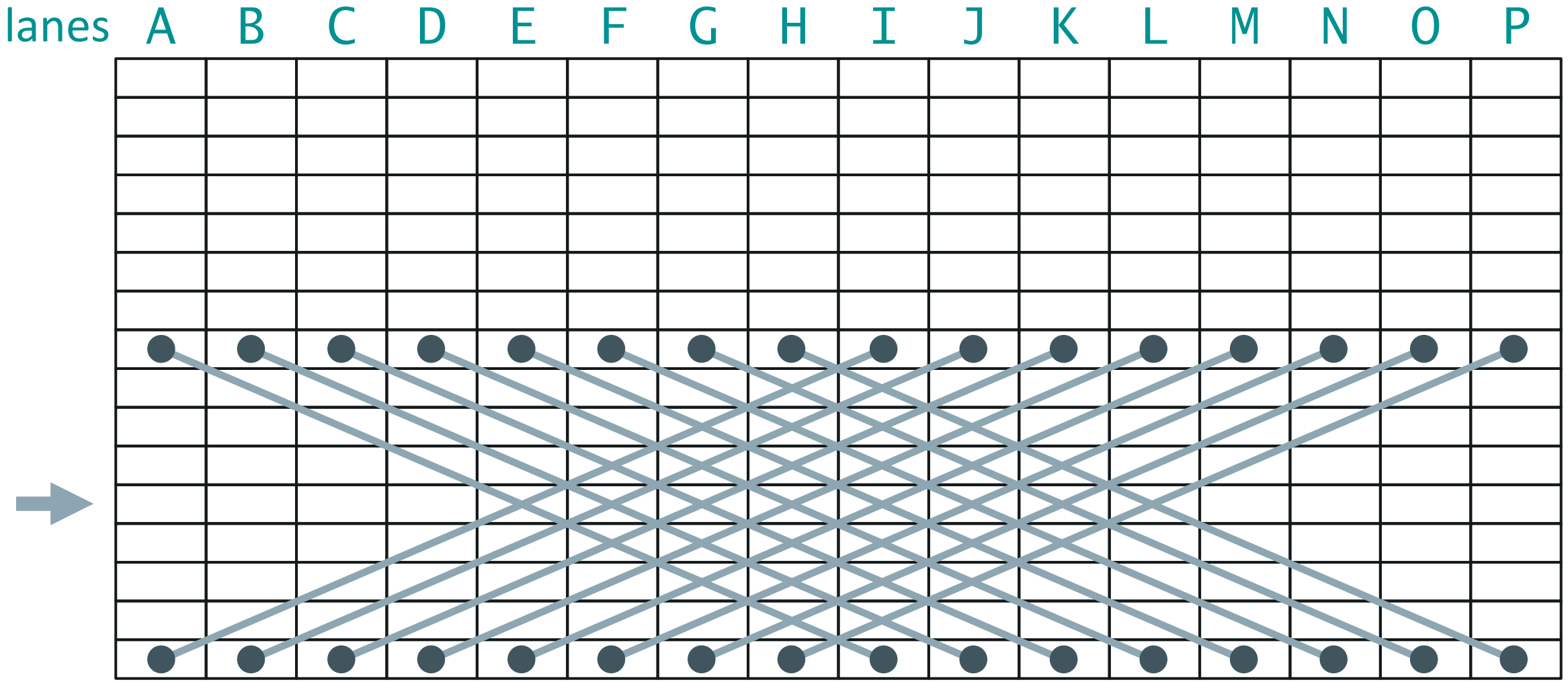
then four __shfl_xor instructions . . .



then two **__shfl_xor** instructions . . .



then one more **__shfl_xor** instruction: total of N–1



Here is the resulting mess:

lanes	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
	A0:0	B0:0	A2:2	B2:2	A4:4	B4:4	A6:6	B6:6	A8:8	B8:8	A10:10	B10:10	A12:12	B12:12	A14:14	B14:14
	A0:1	B0:1	C0:1	D0:1	A4:5	B4:5	C4:5	D4:5	A8:9	B8:9	C8:9	D8:9	A12:13	B12:13	C12:13	D12:13
	C0:0	D0:0	C2:2	D2:2	C4:4	D4:4	C6:6	D6:6	C8:8	D8:8	C10:10	D10:10	C12:12	D12:12	C14:14	D14:14
	A0:3	B0:3	C0:3	D0:3	E0:3	F0:3	G0:3	H0:3	A8:11	B8:11	C8:11	D8:11	E8:11	F8:11	G8:11	H8:11
	E0:0	F0:0	E2:2	F2:2	E4:4	F4:4	E6:6	F6:6	E8:8	F8:8	E10:10	F10:10	E12:12	F12:12	E14:14	F14:14
	E0:1	F0:1	G0:1	H0:1	E4:5	F4:5	G4:5	H4:5	E8:9	F8:9	G8:9	H8:9	E12:13	F12:13	G12:13	H12:13
	G0:0	H0:0	G2:2	H2:2	G4:4	H4:4	G6:6	H6:6	G8:8	H8:8	G10:10	H10:10	G12:12	H12:12	G14:14	H14:14
	A0:7	B0:7	C0:7	D0:7	E0:7	F0:7	G0:7	H0:7	I0:7	J0:7	K0:7	L0:7	M0:7	N0:7	O0:7	P0:7
	I0:0	J0:0	I2:2	J2:2	I4:4	J4:4	I6:6	J6:6	I8:8	J8:8	I10:10	J10:10	I12:12	J12:12	I14:14	J14:14
	I0:1	J0:1	K0:1	L0:1	I4:5	J4:5	K4:5	L4:5	I8:9	J8:9	K8:9	L8:9	I12:13	J12:13	K12:13	L12:13
	K0:0	L0:0	K2:2	L2:2	K4:4	L4:4	K6:6	L6:6	K8:8	L8:8	K10:10	L10:10	K12:12	L12:12	K14:14	L14:14
	I0:3	J0:3	K0:3	L0:3	M0:3	N0:3	O0:3	P0:3	I8:11	J8:11	K8:11	L8:11	M8:11	N8:11	O8:11	P8:11
	M0:0	N0:0	M2:2	N2:2	M4:4	N4:4	M6:6	N6:6	M8:8	N8:8	M10:10	N10:10	M12:12	N12:12	M14:14	N14:14
	M0:1	N0:1	O0:1	P0:1	M4:5	N4:5	O4:5	P4:5	M8:9	N8:9	O8:9	P8:9	M12:13	N12:13	O12:13	P12:13
	O0:0	P0:0	O2:2	P2:2	O4:4	P4:4	O6:6	P6:6	O8:8	P8:8	O10:10	P10:10	O12:12	P12:12	O14:14	P14:14
	A0:15	B0:15	C0:15	D0:15	E0:15	F0:15	G0:15	H0:15	I0:15	J0:15	K0:15	L0:15	M0:15	N0:15	O0:15	P0:15

The overall sums for each lane are there:

lanes	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
	A0:0	B0:0	A2:2	B2:2	A4:4	B4:4	A6:6	B6:6	A8:8	B8:8	A10:10	B10:10	A12:12	B12:12	A14:14	B14:14
	A0:1	B0:1	C0:1	D0:1	A4:5	B4:5	C4:5	D4:5	A8:9	B8:9	C8:9	D8:9	A12:13	B12:13	C12:13	D12:13
	C0:0	D0:0	C2:2	D2:2	C4:4	D4:4	C6:6	D6:6	C8:8	D8:8	C10:10	D10:10	C12:12	D12:12	C14:14	D14:14
	A0:3	B0:3	C0:3	D0:3	E0:3	F0:3	G0:3	H0:3	A8:11	B8:11	C8:11	D8:11	E8:11	F8:11	G8:11	H8:11
	E0:0	F0:0	E2:2	F2:2	E4:4	F4:4	E6:6	F6:6	E8:8	F8:8	E10:10	F10:10	E12:12	F12:12	E14:14	F14:14
	E0:1	F0:1	G0:1	H0:1	E4:5	F4:5	G4:5	H4:5	E8:9	F8:9	G8:9	H8:9	E12:13	F12:13	G12:13	H12:13
	G0:0	H0:0	G2:2	H2:2	G4:4	H4:4	G6:6	H6:6	G8:8	H8:8	G10:10	H10:10	G12:12	H12:12	G14:14	H14:14
	A0:7	B0:7	C0:7	D0:7	E0:7	F0:7	G0:7	H0:7	I0:7	J0:7	K0:7	L0:7	M0:7	N0:7	O0:7	P0:7
	I0:0	J0:0	I2:2	J2:2	I4:4	J4:4	I6:6	J6:6	I8:8	J8:8	I10:10	J10:10	I12:12	J12:12	I14:14	J14:14
	I0:1	J0:1	K0:1	L0:1	I4:5	J4:5	K4:5	L4:5	I8:9	J8:9	K8:9	L8:9	I12:13	J12:13	K12:13	L12:13
	K0:0	L0:0	K2:2	L2:2	K4:4	L4:4	K6:6	L6:6	K8:8	L8:8	K10:10	L10:10	K12:12	L12:12	K14:14	L14:14
	I0:3	J0:3	K0:3	L0:3	M0:3	N0:3	O0:3	P0:3	I8:11	J8:11	K8:11	L8:11	M8:11	N8:11	O8:11	P8:11
	M0:0	N0:0	M2:2	N2:2	M4:4	N4:4	M6:6	N6:6	M8:8	N8:8	M10:10	N10:10	M12:12	N12:12	M14:14	N14:14
	M0:1	N0:1	O0:1	P0:1	M4:5	N4:5	O4:5	P4:5	M8:9	N8:9	O8:9	P8:9	M12:13	N12:13	O12:13	P12:13
	O0:0	P0:0	O2:2	P2:2	O4:4	P4:4	O6:6	P6:6	O8:8	P8:8	O10:10	P10:10	O12:12	P12:12	O14:14	P14:14
→	A0:15	B0:15	C0:15	D0:15	E0:15	F0:15	G0:15	H0:15	I0:15	J0:15	K0:15	L0:15	M0:15	N0:15	O0:15	P0:15

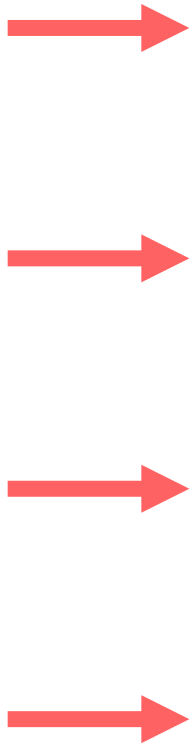
The roots of all **N** incremental trees are also there . . .

lanes	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
	A0:0	B0:0	A2:2	B2:2	A4:4	B4:4	A6:6	B6:6	A8:8	B8:8	A10:10	B10:10	A12:12	B12:12	A14:14	B14:14
	A0:1	B0:1	C0:1	D0:1	A4:5	B4:5	C4:5	D4:5	A8:9	B8:9	C8:9	D8:9	A12:13	B12:13	C12:13	D12:13
	C0:0	D0:0	C2:2	D2:2	C4:4	D4:4	C6:6	D6:6	C8:8	D8:8	C10:10	D10:10	C12:12	D12:12	C14:14	D14:14
	A0:3	B0:3	C0:3	D0:3	E0:3	F0:3	G0:3	H0:3	A8:11	B8:11	C8:11	D8:11	E8:11	F8:11	G8:11	H8:11
	E0:0	F0:0	E2:2	F2:2	E4:4	F4:4	E6:6	F6:6	E8:8	F8:8	E10:10	F10:10	E12:12	F12:12	E14:14	F14:14
	E0:1	F0:1	G0:1	H0:1	E4:5	F4:5	G4:5	H4:5	E8:9	F8:9	G8:9	H8:9	E12:13	F12:13	G12:13	H12:13
	G0:0	H0:0	G2:2	H2:2	G4:4	H4:4	G6:6	H6:6	G8:8	H8:8	G10:10	H10:10	G12:12	H12:12	G14:14	H14:14
→	A0:7	B0:7	C0:7	D0:7	E0:7	F0:7	G0:7	H0:7	I0:7	J0:7	K0:7	L0:7	M0:7	N0:7	O0:7	P0:7
	I0:0	J0:0	I2:2	J2:2	I4:4	J4:4	I6:6	J6:6	I8:8	J8:8	I10:10	J10:10	I12:12	J12:12	I14:14	J14:14
	I0:1	J0:1	K0:1	L0:1	I4:5	J4:5	K4:5	L4:5	I8:9	J8:9	K8:9	L8:9	I12:13	J12:13	K12:13	L12:13
	K0:0	L0:0	K2:2	L2:2	K4:4	L4:4	K6:6	L6:6	K8:8	L8:8	K10:10	L10:10	K12:12	L12:12	K14:14	L14:14
	I0:3	J0:3	K0:3	L0:3	M0:3	N0:3	O0:3	P0:3	I8:11	J8:11	K8:11	L8:11	M8:11	N8:11	O8:11	P8:11
	M0:0	N0:0	M2:2	N2:2	M4:4	N4:4	M6:6	N6:6	M8:8	N8:8	M10:10	N10:10	M12:12	N12:12	M14:14	N14:14
	M0:1	N0:1	O0:1	P0:1	M4:5	N4:5	O4:5	P4:5	M8:9	N8:9	O8:9	P8:9	M12:13	N12:13	O12:13	P12:13
	O0:0	P0:0	O2:2	P2:2	O4:4	P4:4	O6:6	P6:6	O8:8	P8:8	O10:10	P10:10	O12:12	P12:12	O14:14	P14:14
	A0:15	B0:15	C0:15	D0:15	E0:15	F0:15	G0:15	H0:15	I0:15	J0:15	K0:15	L0:15	M0:15	N0:15	O0:15	P0:15

and also the second-level tree nodes . . .

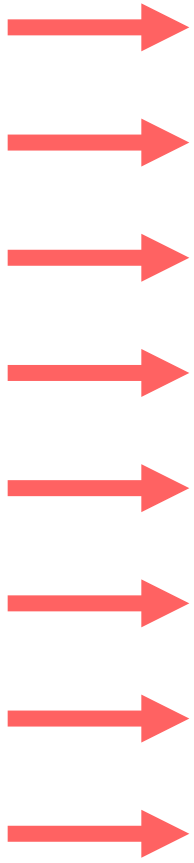
lanes	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
	A0:0	B0:0	A2:2	B2:2	A4:4	B4:4	A6:6	B6:6	A8:8	B8:8	A10:10	B10:10	A12:12	B12:12	A14:14	B14:14
	A0:1	B0:1	C0:1	D0:1	A4:5	B4:5	C4:5	D4:5	A8:9	B8:9	C8:9	D8:9	A12:13	B12:13	C12:13	D12:13
	C0:0	D0:0	C2:2	D2:2	C4:4	D4:4	C6:6	D6:6	C8:8	D8:8	C10:10	D10:10	C12:12	D12:12	C14:14	D14:14
	A0:3	B0:3	C0:3	D0:3	E0:3	F0:3	G0:3	H0:3	A8:11	B8:11	C8:11	D8:11	E8:11	F8:11	G8:11	H8:11
	E0:0	F0:0	E2:2	F2:2	E4:4	F4:4	E6:6	F6:6	E8:8	F8:8	E10:10	F10:10	E12:12	F12:12	E14:14	F14:14
	E0:1	F0:1	G0:1	H0:1	E4:5	F4:5	G4:5	H4:5	E8:9	F8:9	G8:9	H8:9	E12:13	F12:13	G12:13	H12:13
	G0:0	H0:0	G2:2	H2:2	G4:4	H4:4	G6:6	H6:6	G8:8	H8:8	G10:10	H10:10	G12:12	H12:12	G14:14	H14:14
	A0:7	B0:7	C0:7	D0:7	E0:7	F0:7	G0:7	H0:7	I0:7	J0:7	K0:7	L0:7	M0:7	N0:7	O0:7	P0:7
	I0:0	J0:0	I2:2	J2:2	I4:4	J4:4	I6:6	J6:6	I8:8	J8:8	I10:10	J10:10	I12:12	J12:12	I14:14	J14:14
	I0:1	J0:1	K0:1	L0:1	I4:5	J4:5	K4:5	L4:5	I8:9	J8:9	K8:9	L8:9	I12:13	J12:13	K12:13	L12:13
	K0:0	L0:0	K2:2	L2:2	K4:4	L4:4	K6:6	L6:6	K8:8	L8:8	K10:10	L10:10	K12:12	L12:12	K14:14	L14:14
	I0:3	J0:3	K0:3	L0:3	M0:3	N0:3	O0:3	P0:3	I8:11	J8:11	K8:11	L8:11	M8:11	N8:11	O8:11	P8:11
	M0:0	N0:0	M2:2	N2:2	M4:4	N4:4	M6:6	N6:6	M8:8	N8:8	M10:10	N10:10	M12:12	N12:12	M14:14	N14:14
	M0:1	N0:1	O0:1	P0:1	M4:5	N4:5	O4:5	P4:5	M8:9	N8:9	O8:9	P8:9	M12:13	N12:13	O12:13	P12:13
	O0:0	P0:0	O2:2	P2:2	O4:4	P4:4	O6:6	P6:6	O8:8	P8:8	O10:10	P10:10	O12:12	P12:12	O14:14	P14:14
	A0:15	B0:15	C0:15	D0:15	E0:15	F0:15	G0:15	H0:15	I0:15	J0:15	K0:15	L0:15	M0:15	N0:15	O0:15	P0:15

and the third-level tree nodes . . .



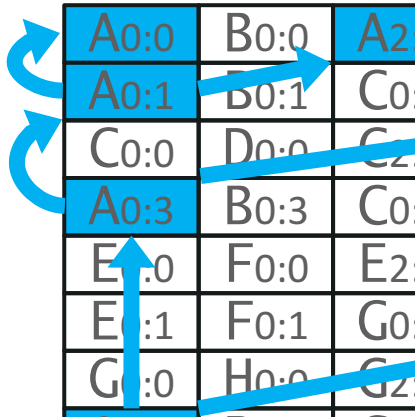
lanes	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
	A0:0	B0:0	A2:2	B2:2	A4:4	B4:4	A6:6	B6:6	A8:8	B8:8	A10:10	B10:10	A12:12	B12:12	A14:14	B14:14
→	A0:1	B0:1	C0:1	D0:1	A4:5	B4:5	C4:5	D4:5	A8:9	B8:9	C8:9	D8:9	A12:13	B12:13	C12:13	D12:13
	C0:0	D0:0	C2:2	D2:2	C4:4	D4:4	C6:6	D6:6	C8:8	D8:8	C10:10	D10:10	C12:12	D12:12	C14:14	D14:14
	A0:3	B0:3	C0:3	D0:3	E0:3	F0:3	G0:3	H0:3	A8:11	B8:11	C8:11	D8:11	E8:11	F8:11	G8:11	H8:11
	E0:0	F0:0	E2:2	F2:2	E4:4	F4:4	E6:6	F6:6	E8:8	F8:8	E10:10	F10:10	E12:12	F12:12	E14:14	F14:14
→	E0:1	F0:1	G0:1	H0:1	E4:5	F4:5	G4:5	H4:5	E8:9	F8:9	G8:9	H8:9	E12:13	F12:13	G12:13	H12:13
	G0:0	H0:0	G2:2	H2:2	G4:4	H4:4	G6:6	H6:6	G8:8	H8:8	G10:10	H10:10	G12:12	H12:12	G14:14	H14:14
	A0:7	B0:7	C0:7	D0:7	E0:7	F0:7	G0:7	H0:7	I0:7	J0:7	K0:7	L0:7	M0:7	N0:7	O0:7	P0:7
	I0:0	J0:0	I2:2	J2:2	I4:4	J4:4	I6:6	J6:6	I8:8	J8:8	I10:10	J10:10	I12:12	J12:12	I14:14	J14:14
→	I0:1	J0:1	K0:1	L0:1	I4:5	J4:5	K4:5	L4:5	I8:9	J8:9	K8:9	L8:9	I12:13	J12:13	K12:13	L12:13
	K0:0	L0:0	K2:2	L2:2	K4:4	L4:4	K6:6	L6:6	K8:8	L8:8	K10:10	L10:10	K12:12	L12:12	K14:14	L14:14
	I0:3	J0:3	K0:3	L0:3	M0:3	N0:3	O0:3	P0:3	I8:11	J8:11	K8:11	L8:11	M8:11	N8:11	O8:11	P8:11
	M0:0	N0:0	M2:2	N2:2	M4:4	N4:4	M6:6	N6:6	M8:8	N8:8	M10:10	N10:10	M12:12	N12:12	M14:14	N14:14
→	M0:1	N0:1	O0:1	P0:1	M4:5	N4:5	O4:5	P4:5	M8:9	N8:9	O8:9	P8:9	M12:13	N12:13	O12:13	P12:13
	O0:0	P0:0	O2:2	P2:2	O4:4	P4:4	O6:6	P6:6	O8:8	P8:8	O10:10	P10:10	O12:12	P12:12	O14:14	P14:14
	A0:15	B0:15	C0:15	D0:15	E0:15	F0:15	G0:15	H0:15	I0:15	J0:15	K0:15	L0:15	M0:15	N0:15	O0:15	P0:15

all the way down to the bottom!



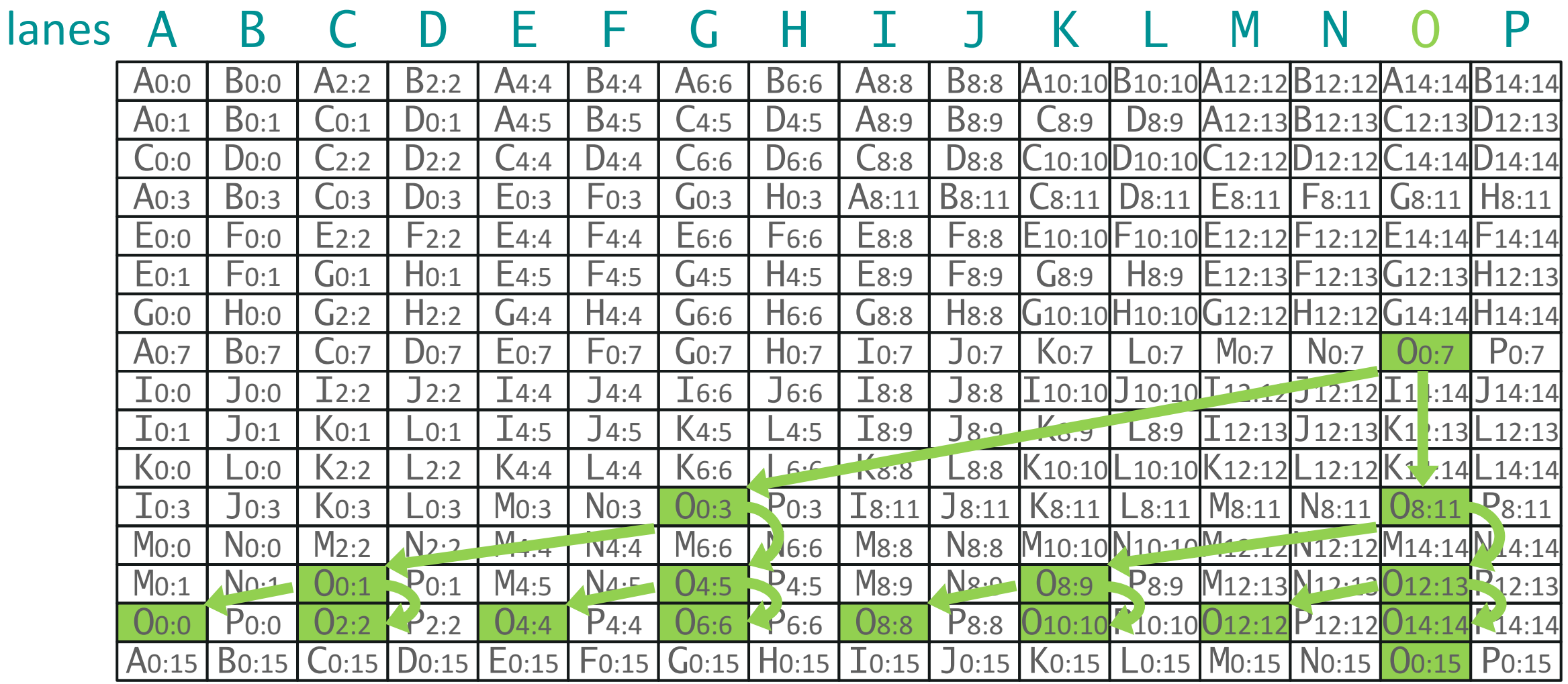
lanes	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
	A0:0	B0:0	A2:2	B2:2	A4:4	B4:4	A6:6	B6:6	A8:8	B8:8	A10:10	B10:10	A12:12	B12:12	A14:14	B14:14
	A0:1	B0:1	C0:1	D0:1	A4:5	B4:5	C4:5	D4:5	A8:9	B8:9	C8:9	D8:9	A12:13	B12:13	C12:13	D12:13
	C0:0	D0:0	C2:2	D2:2	C4:4	D4:4	C6:6	D6:6	C8:8	D8:8	C10:10	D10:10	C12:12	D12:12	C14:14	D14:14
	A0:3	B0:3	C0:3	D0:3	E0:3	F0:3	G0:3	H0:3	A8:11	B8:11	C8:11	D8:11	E8:11	F8:11	G8:11	H8:11
	E0:0	F0:0	E2:2	F2:2	E4:4	F4:4	E6:6	F6:6	E8:8	F8:8	E10:10	F10:10	E12:12	F12:12	E14:14	F14:14
	E0:1	F0:1	G0:1	H0:1	E4:5	F4:5	G4:5	H4:5	E8:9	F8:9	G8:9	H8:9	E12:13	F12:13	G12:13	H12:13
	G0:0	H0:0	G2:2	H2:2	G4:4	H4:4	G6:6	H6:6	G8:8	H8:8	G10:10	H10:10	G12:12	H12:12	G14:14	H14:14
	A0:7	B0:7	C0:7	D0:7	E0:7	F0:7	G0:7	H0:7	I0:7	J0:7	K0:7	L0:7	M0:7	N0:7	O0:7	P0:7
	I0:0	J0:0	I2:2	J2:2	I4:4	J4:4	I6:6	J6:6	I8:8	J8:8	I10:10	J10:10	I12:12	J12:12	I14:14	J14:14
	I0:1	J0:1	K0:1	L0:1	I4:5	J4:5	K4:5	L4:5	I8:9	J8:9	K8:9	L8:9	I12:13	J12:13	K12:13	L12:13
	K0:0	L0:0	K2:2	L2:2	K4:4	L4:4	K6:6	L6:6	K8:8	L8:8	K10:10	L10:10	K12:12	L12:12	K14:14	L14:14
	I0:3	J0:3	K0:3	L0:3	M0:3	N0:3	O0:3	P0:3	I8:11	J8:11	K8:11	L8:11	M8:11	N8:11	O8:11	P8:11
	M0:0	N0:0	M2:2	N2:2	M4:4	N4:4	M6:6	N6:6	M8:8	N8:8	M10:10	N10:10	M12:12	N12:12	M14:14	N14:14
	M0:1	N0:1	O0:1	P0:1	M4:5	N4:5	O4:5	P4:5	M8:9	N8:9	O8:9	P8:9	M12:13	N12:13	O12:13	P12:13
	O0:0	P0:0	O2:2	P2:2	O4:4	P4:4	O6:6	P6:6	O8:8	P8:8	O10:10	P10:10	O12:12	P12:12	O14:14	P14:14
	A0:15	B0:15	C0:15	D0:15	E0:15	F0:15	G0:15	H0:15	I0:15	J0:15	K0:15	L0:15	M0:15	N0:15	O0:15	P0:15

The Tree and Total Sum for Lane A

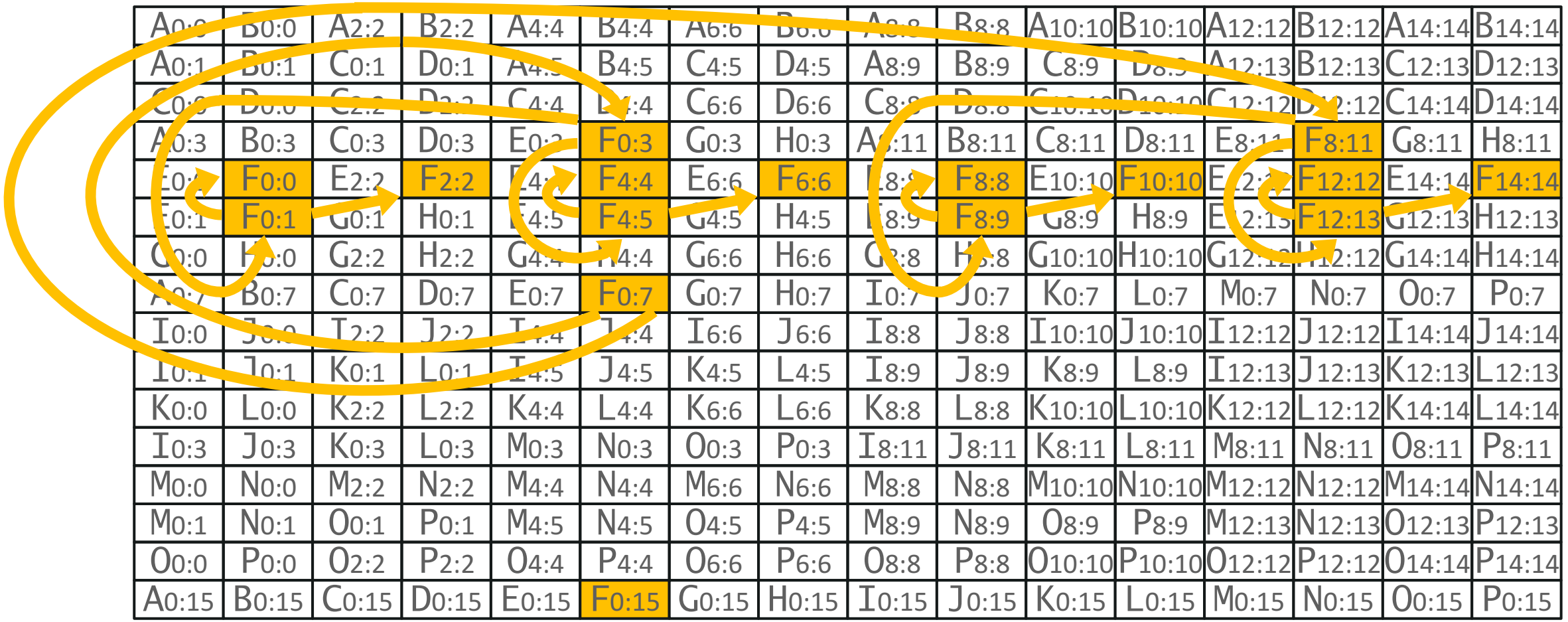


lanes	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
	A0:0	B0:0	A2:2	B2:2	A4:4	B4:4	A6:6	B6:6	A8:8	B8:8	A10:10	B10:10	A12:12	B12:12	A14:14	B14:14
	A0:1	B0:1	C0:1	D0:1	A4:5	B4:5	C4:5	D4:5	A8:9	B8:9	C8:9	D8:9	A12:13	B12:13	C12:13	D12:13
	C0:0	D0:0	C2:2	D2:2	C4:4	D4:4	C6:6	D6:6	C8:8	D8:8	C10:10	D10:10	C12:12	D12:12	C14:14	D14:14
	A0:3	B0:3	C0:3	D0:3	E0:3	F0:3	G0:3	H0:3	A8:11	B8:11	C8:11	D8:11	E8:11	F8:11	G8:11	H8:11
	E0:0	F0:0	E2:2	F2:2	E4:4	F4:4	E6:6	F6:6	E8:8	F8:8	E10:10	F10:10	E12:12	F12:12	E14:14	F14:14
	E0:1	F0:1	G0:1	H0:1	E4:5	F4:5	G4:5	H4:5	E8:9	F8:9	G8:9	H8:9	E12:13	F12:13	G12:13	H12:13
	G0:0	H0:0	G2:2	H2:2	G4:4	H4:4	G6:6	H6:6	G8:8	H8:8	G10:10	H10:10	G12:12	H12:12	G14:14	H14:14
	A0:7	B0:7	C0:7	D0:7	E0:7	F0:7	G0:7	H0:7	I0:7	J0:7	K0:7	L0:7	M0:7	N0:7	O0:7	P0:7
	I0:0	J0:0	I2:2	J2:2	I4:4	J4:4	I6:6	J6:6	I8:8	J8:8	I10:10	J10:10	I12:12	J12:12	I14:14	J14:14
	I0:1	J0:1	K0:1	L0:1	I4:5	J4:5	K4:5	L4:5	I8:9	J8:9	K8:9	L8:9	I12:13	J12:13	K12:13	L12:13
	K0:0	L0:0	K2:2	L2:2	K4:4	L4:4	K6:6	L6:6	K8:8	L8:8	K10:10	L10:10	K12:12	L12:12	K14:14	L14:14
	I0:3	J0:3	K0:3	L0:3	M0:3	N0:3	O0:3	P0:3	I8:11	J8:11	K8:11	L8:11	M8:11	N8:11	O8:11	P8:11
	M0:0	N0:0	M2:2	N2:2	M4:4	N4:4	M6:6	N6:6	M8:8	N8:8	M10:10	N10:10	M12:12	N12:12	M14:14	N14:14
	M0:1	N0:1	O0:1	P0:1	M4:5	N4:5	O4:5	P4:5	M8:9	N8:9	O8:9	P8:9	M12:13	N12:13	O12:13	P12:13
	O0:0	P0:0	O2:2	P2:2	O4:4	P4:4	O6:6	P6:6	O8:8	P8:8	O10:10	P10:10	O12:12	P12:12	O14:14	P14:14
	A0:15	B0:15	C0:15	D0:15	E0:15	F0:15	G0:15	H0:15	I0:15	J0:15	K0:15	L0:15	M0:15	N0:15	O0:15	P0:15

The Tree and Total Sum for Lane 0



The Tree and Total Sum for Lane F



lanes	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
	A0:0	B0:0	A2:2	B2:2	A4:4	B4:4	A6:6	B6:6	A8:8	B8:8	A10:10	B10:10	A12:12	B12:12	A14:14	B14:14
	A0:1	B0:1	C0:1	D0:1	A4:5	B4:5	C4:5	D4:5	A8:9	B8:9	C8:9	D8:9	A12:13	B12:13	C12:13	D12:13
	C0:0	D0:0	C2:2	D2:2	C4:4	D4:4	C6:6	D6:6	C8:8	D8:8	C10:10	D10:10	C12:12	D12:12	C14:14	D14:14
	A0:3	B0:3	C0:3	D0:3	E0:3	F0:3	G0:3	H0:3	A8:11	B8:11	C8:11	D8:11	E8:11	F8:11	G8:11	H8:11
	E0:0	F0:0	E2:2	F2:2	E4:4	F4:4	E6:6	F6:6	E8:8	F8:8	E10:10	F10:10	E12:12	F12:12	E14:14	F14:14
	E0:1	F0:1	G0:1	H0:1	E4:5	F4:5	G4:5	H4:5	E8:9	F8:9	G8:9	H8:9	E12:13	F12:13	G12:13	H12:13
	G0:0	H0:0	G2:2	H2:2	G4:4	H4:4	G6:6	H6:6	G8:8	H8:8	G10:10	H10:10	G12:12	H12:12	G14:14	H14:14
	A0:7	B0:7	C0:7	D0:7	E0:7	F0:7	G0:7	H0:7	I0:7	J0:7	K0:7	L0:7	M0:7	N0:7	O0:7	P0:7
	I0:0	J0:0	I2:2	J2:2	I4:4	J4:4	I6:6	J6:6	I8:8	J8:8	I10:10	J10:10	I12:12	J12:12	I14:14	J14:14
	I0:1	J0:1	K0:1	L0:1	I4:5	J4:5	K4:5	L4:5	I8:9	J8:9	K8:9	L8:9	I12:13	J12:13	K12:13	L12:13
	K0:0	L0:0	K2:2	L2:2	K4:4	L4:4	K6:6	L6:6	K8:8	L8:8	K10:10	L10:10	K12:12	L12:12	K14:14	L14:14
	I0:3	J0:3	K0:3	L0:3	M0:3	N0:3	O0:3	P0:3	I8:11	J8:11	K8:11	L8:11	M8:11	N8:11	O8:11	P8:11
	M0:0	N0:0	M2:2	N2:2	M4:4	N4:4	M6:6	N6:6	M8:8	N8:8	M10:10	N10:10	M12:12	N12:12	M14:14	N14:14
	M0:1	N0:1	O0:1	P0:1	M4:5	N4:5	O4:5	P4:5	M8:9	N8:9	O8:9	P8:9	M12:13	N12:13	O12:13	P12:13
	O0:0	P0:0	O2:2	P2:2	O4:4	P4:4	O6:6	P6:6	O8:8	P8:8	O10:10	P10:10	O12:12	P12:12	O14:14	P14:14
	A0:15	B0:15	C0:15	D0:15	E0:15	F0:15	G0:15	H0:15	I0:15	J0:15	K0:15	L0:15	M0:15	N0:15	O0:15	P0:15

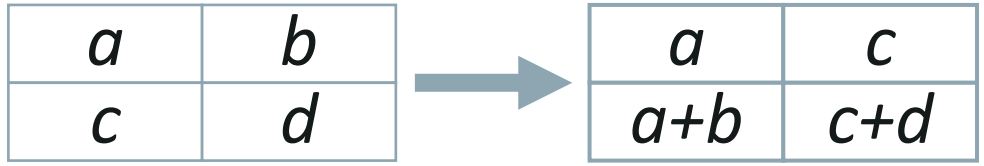
Code to Create the Butterfly-Patterned Partial-Sums Table

```
int k = threadIdx.x & 0x1f;                /* lane ID */
for (int b = 0; b < N; b += b) {           /* 1, 2, 4, 8, … */
  for (int j = 0; j < (N>>(b+1)); j++) {
    d = (((j << 1) + 1) << b) - 1;
    h = (k & (1 << b)) ? a[d] : a[d+(1 << b)];
    v = __shfl_xor(h, 1 << b);
    if (k & (1 << b)) a[d] = v;
    else a[d + (1 << b)] = v;
    a[d + (1 << b)] += a[d];
    p[d] = a[d];
  }
}
p[N-1] = a[N-1];
```

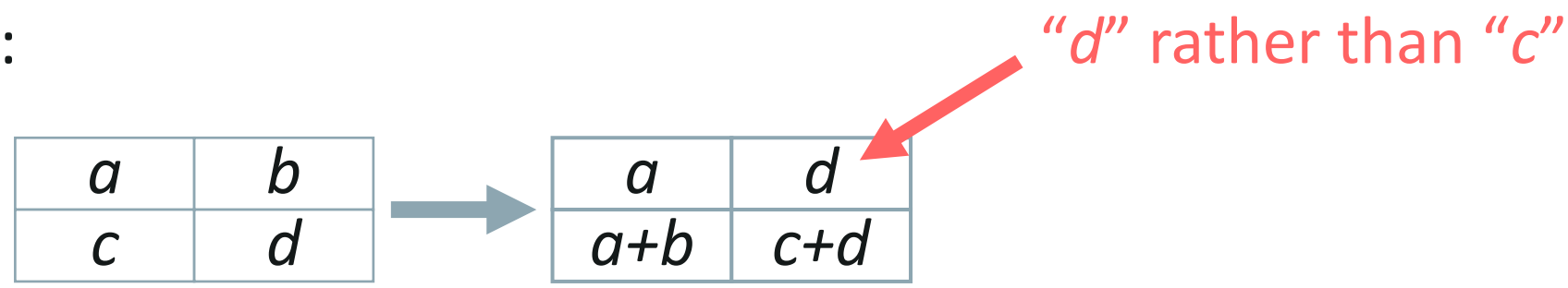
TWO conditional moves and one add

One More Clever Tweak

- Instead of this computation pattern:



we use this one:



- Why? The code is faster! (By maybe another 15% or so!)

The Big Idea: Creating N Such Trees in One Pass

```
int k = threadIdx.x & 0x1f;                /* lane ID */
for (int b = 0; b < N; b += b) {           /* 1, 2, 4, 8, … */
  for (int j = 0; j < (N>>(b+1)); j++) {
    d = (((j << 1) + 1) << b) - 1;
    h = (k & (1 << b)) ? a[d] : a[d+(1 << b)];
    v = __shfl_xor(h, 1 << b);
    if (k & (1 << b)) a[d] = a[d + (1 << b)];
    a[d + (1 << b)] = a[d] + v;
    p[d] = a[d];
  }
}
p[N-1] = a[N-1];
```

ONE conditional move
and one add

This Computes a Different Pattern

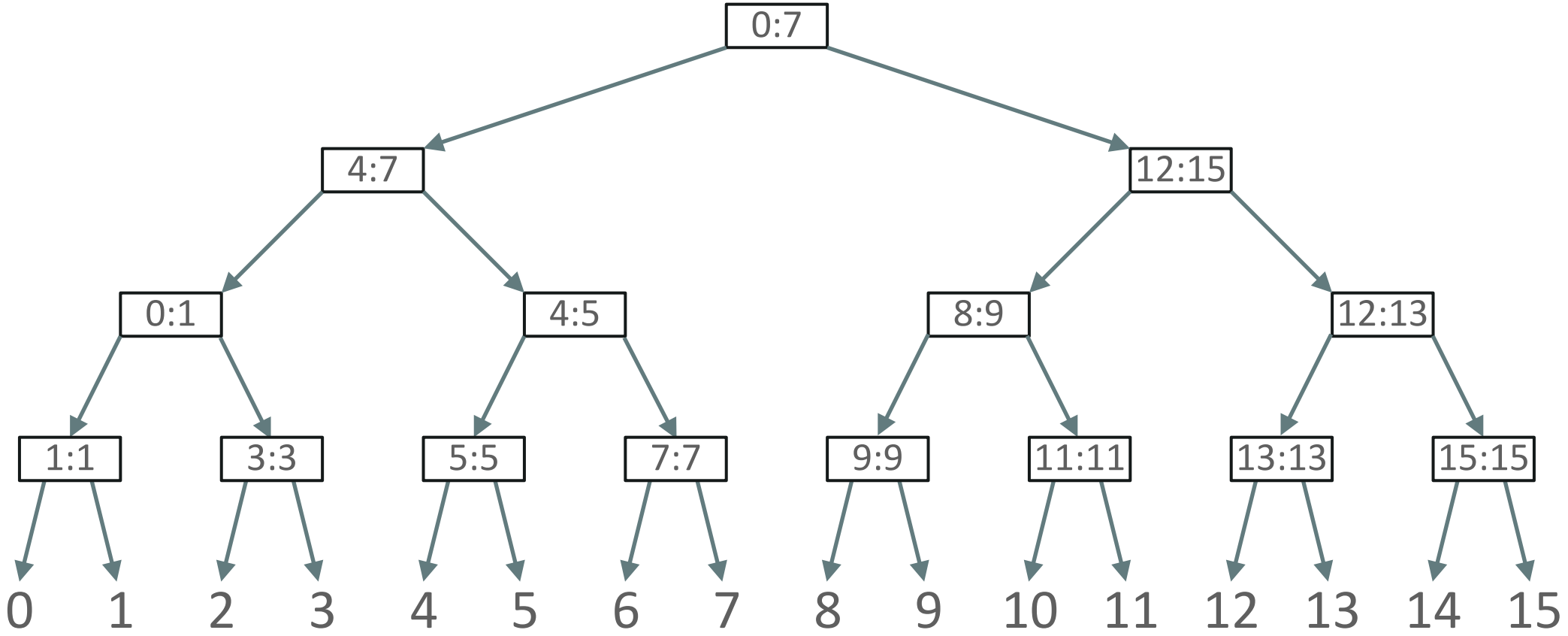
lanes	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
	A0:0	B1:1	A2:2	B3:3	A4:4	B5:5	A6:6	B7:7	A8:8	B9:9	A10:10	B11:11	A12:12	B13:13	A14:14	B15:15
	A0:1	B0:1	C2:3	D2:3	A4:5	B4:5	C6:7	D6:7	A8:9	B8:9	C10:11	D10:11	A12:13	B12:13	C14:15	D14:15
	C0:0	D1:1	C2:2	D3:3	C4:4	D5:5	C6:6	D7:7	C8:8	D9:9	C10:10	D11:11	C12:12	D13:13	C14:14	D15:15
	A0:3	B0:3	C0:3	D0:3	E4:7	F4:7	G4:7	H4:7	A8:11	B8:11	C8:11	D8:11	E12:15	F12:15	G12:15	H12:15
	E0:0	F1:1	E2:2	F3:3	E4:4	F5:5	E6:6	F7:7	E8:8	F9:9	E10:10	F11:11	E12:12	F13:13	E14:14	F15:15
	E0:1	F0:1	G2:3	H2:3	E4:5	F4:5	G6:7	H6:7	E8:9	F8:9	G10:11	H10:11	E12:13	F12:13	G14:15	H14:15
	G0:0	H1:1	G2:2	H3:3	G4:4	H5:5	G6:6	H7:7	G8:8	H9:9	G10:10	H11:11	G12:12	H13:13	G14:14	H15:15
	A0:7	B0:7	C0:7	D0:7	E0:7	F0:7	G0:7	H0:7	I8:15	J8:15	K8:15	L8:15	M8:15	N8:15	O8:15	P8:15
	I0:0	J1:1	I2:2	J3:3	I4:4	J5:5	I6:6	J7:7	I8:8	J9:9	I10:10	J11:11	I12:12	J13:13	I14:14	J15:15
	I0:1	J0:1	K2:3	L2:3	I4:5	J4:5	K6:7	L6:7	I8:9	J8:9	K10:11	L10:11	I12:13	J12:13	K14:15	L14:15
	K0:0	L1:1	K2:2	L3:3	K4:4	L5:5	K6:6	L7:7	K8:8	L9:9	K10:10	L11:11	K12:12	L13:13	K14:14	L15:15
	I0:3	J0:3	K0:3	L0:3	M4:7	N4:7	O4:7	P4:7	I8:11	J8:11	K8:11	L8:11	M12:15	N12:15	O12:15	P12:15
	M0:0	N1:1	M2:2	N3:3	M4:4	N5:5	M6:6	N7:7	M8:8	N9:9	M10:10	N11:11	M12:12	N13:13	M14:14	N15:15
	M0:1	N0:1	O2:3	P2:3	M4:5	N4:5	O6:7	P6:7	M8:9	N8:9	O10:11	P10:11	M12:13	N12:13	O14:15	P14:15
	O0:0	P1:1	O2:2	P3:3	O4:4	P5:5	O6:6	P7:7	O8:8	P9:9	O10:10	P11:11	O12:12	P13:13	O14:14	P15:15
	A0:15	B0:15	C0:15	D0:15	E0:15	F0:15	G0:15	H0:15	I0:15	J0:15	K0:15	L0:15	M0:15	N0:15	O0:15	P0:15

Seraching an “Add/Subtract Search Tree” (1 of 4)

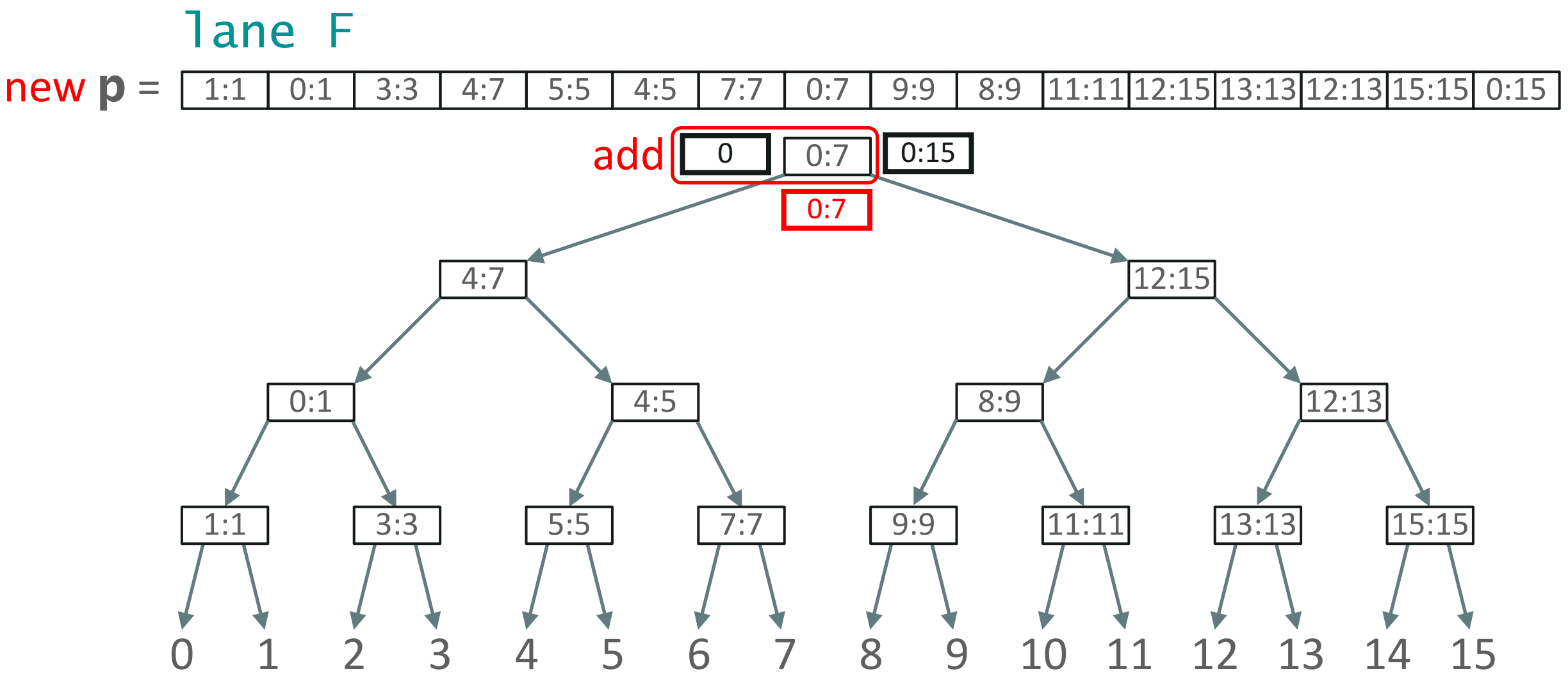
lane F

new **p** =

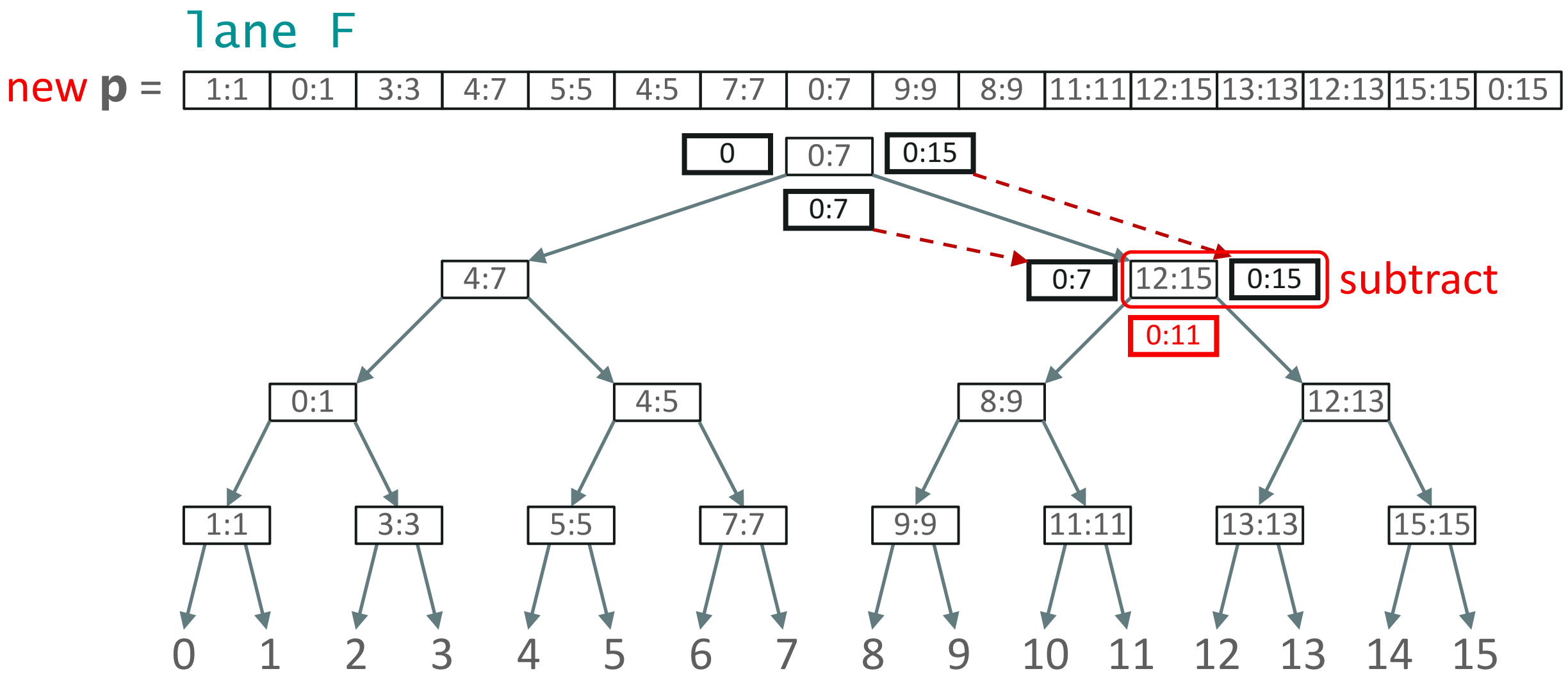
1:1	0:1	3:3	4:7	5:5	4:5	7:7	0:7	9:9	8:9	11:11	12:15	13:13	12:13	15:15	0:15



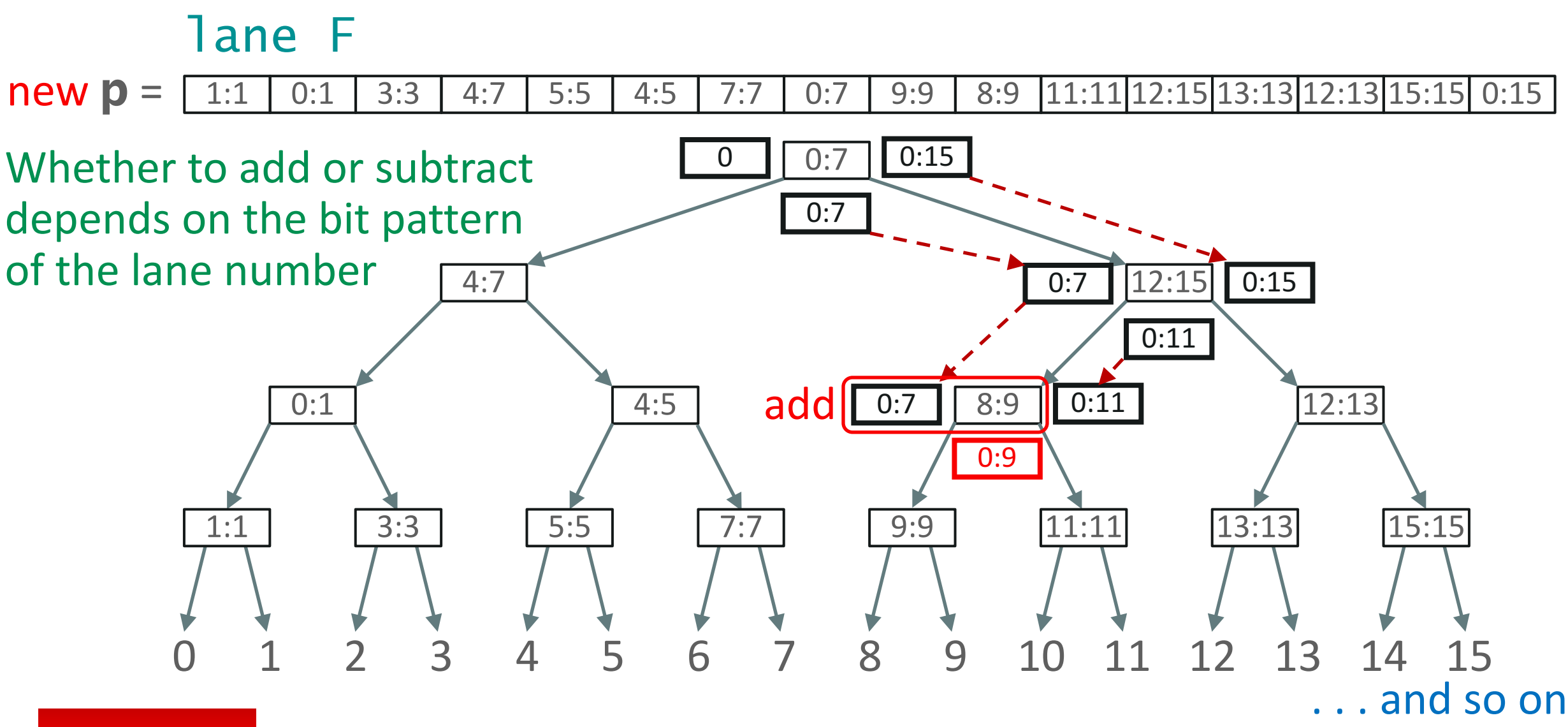
Seraching an “Add/Subtract Search Tree” (2 of 4)



Seraching an “Add/Subtract Search Tree” (3 of 4)



Seraching an “Add/Subtract Search Tree” (4 of 4)



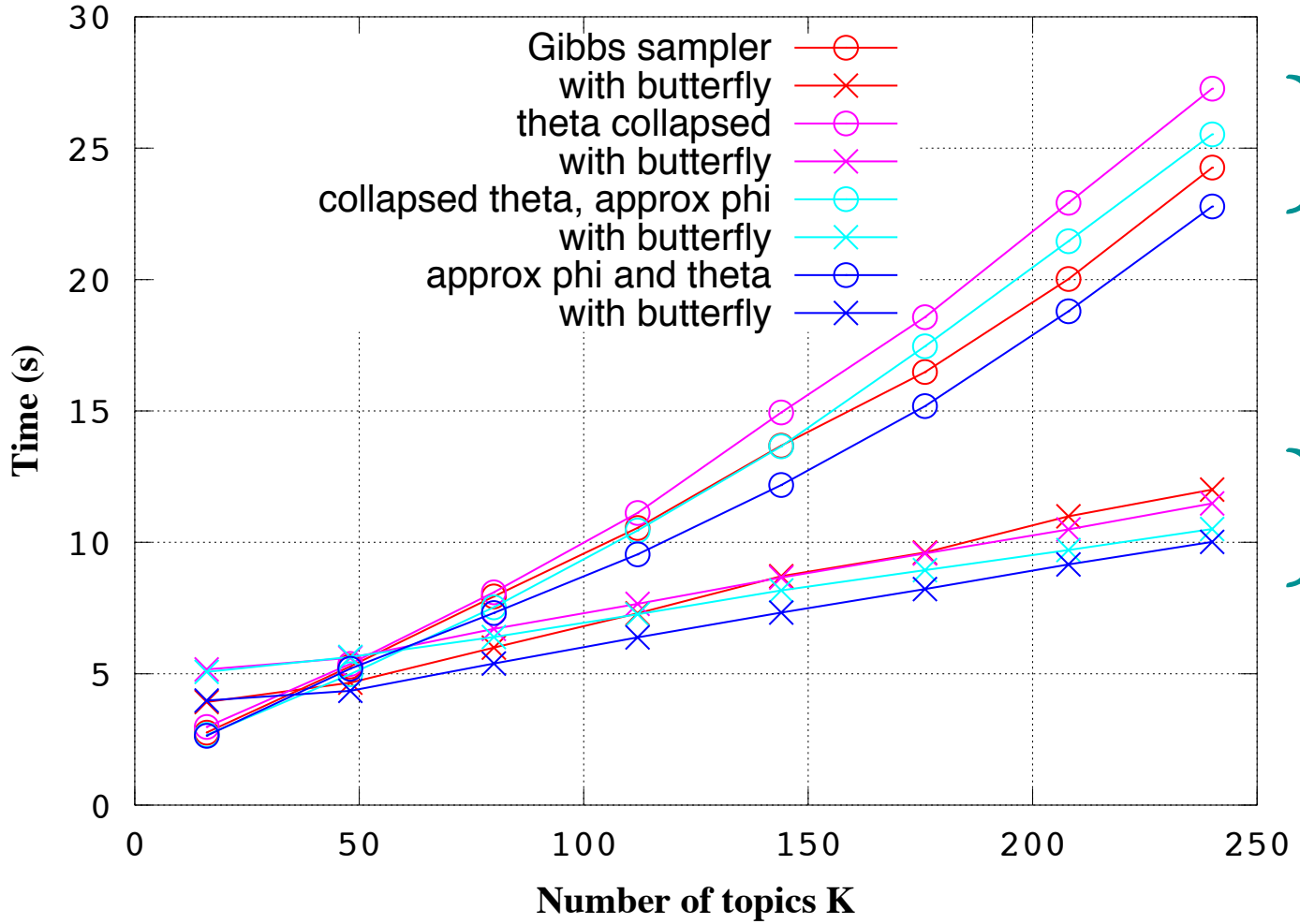
What If **N** Is Not Equal to the Number of Warp Lanes?

- Divide **a** into blocks of size 32 (if **N** mod 32 > 0, there is also a "remnant")
  - If remnant size is larger than 8, pad the table with 0's so its length is a multiple of 32
- Turn each block of 32 into a butterfly-patterned partial-sums table
  - The last row of each block is the sum of all elements in that block for that lane
- Replace the remnant with its running totals, then replace the last rows of of all blocks (including the remnant) with their running totals
- When searching the table:
  - Perform an initial binary search within the "last rows"; this identifies a block
  - If the block is the remnant, search it linearly
  - If the block is not the remnant, use the special add-subtract binary search

Recap

- In our application, memory access (fetching data for relative probabilities) was the performance bottleneck
- Transposed access solves this problem, but now lanes must exchange data
- With per-lane register indexing, full transpose would take only 32 shuffles (assuming 32 lanes), but current architecture does not support that
- Without per-lane register indexing, full transpose using the butterfly (**`__shfl_xor`**) technique takes 80 shuffles
- But using 31 shuffles, we can construct a butterfly-patterned partial-sums tables that contains all 32 incremental (or add-subtract) trees
- Binary search gets more complicated, but the trade-off is worth it

Measurements on Our Machine-Learning Application (2015)

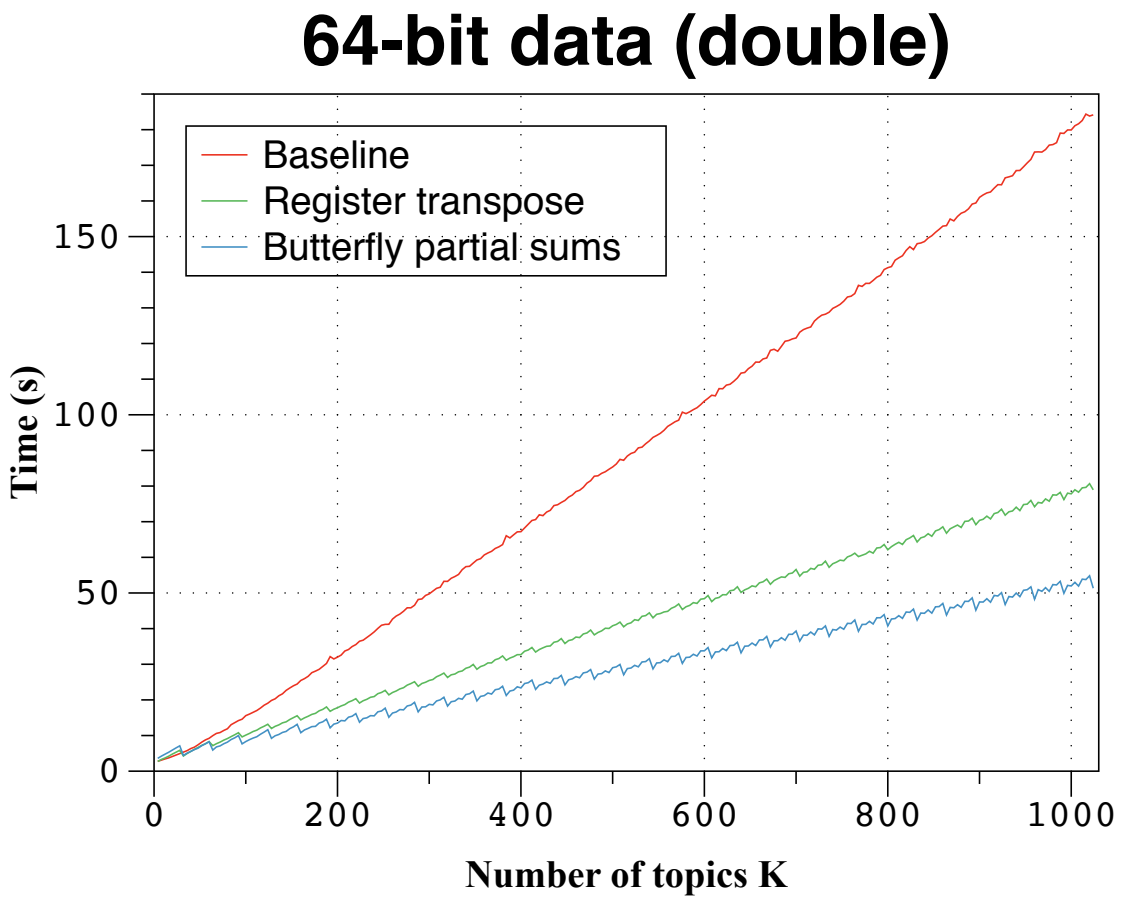


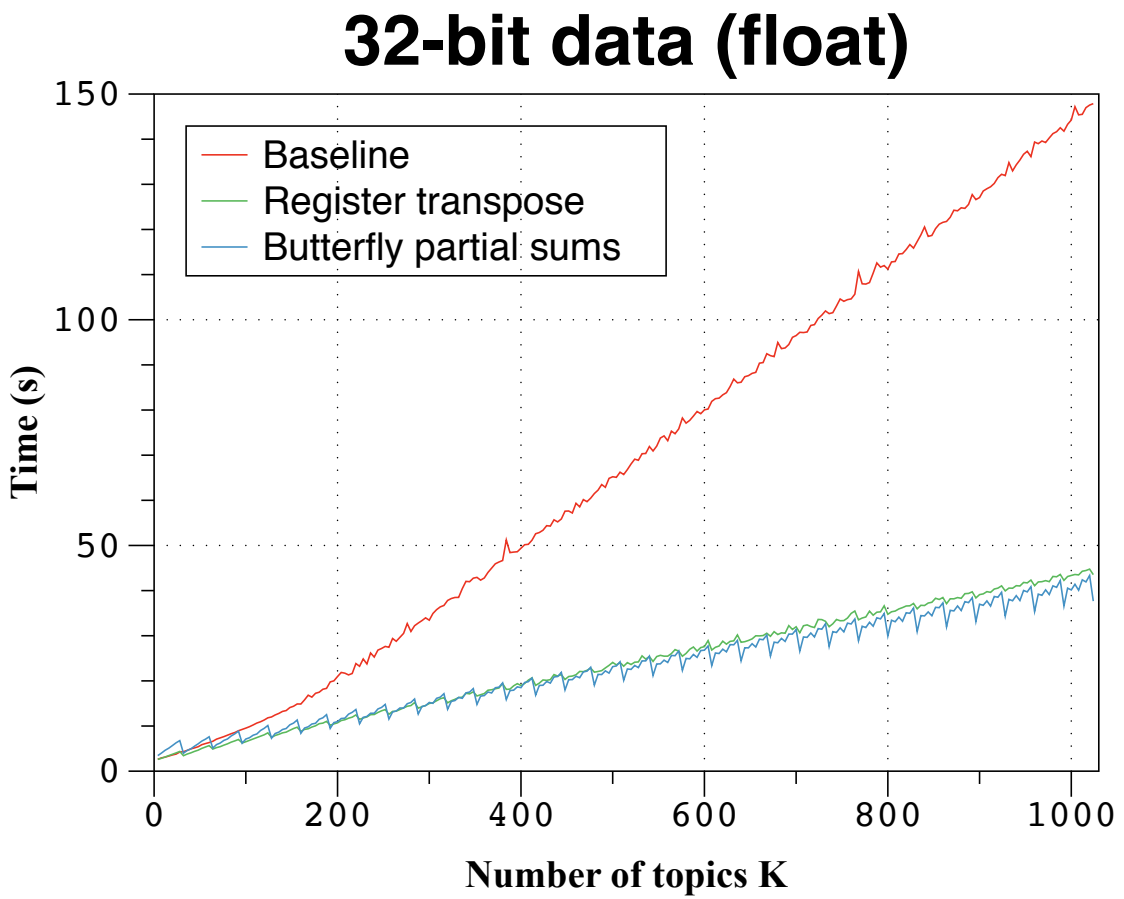
using normal memory accesses and complete sum-prefix tables

using transposed memory accesses and butterfly-patterned partial-sums tables: faster when K > 80; more than twice as fast when K > 200

Coded in CUDA 6.0 for a single NVIDIA Titan Black GPU. Wikipedia-based dataset with number of documents M = 43556, vocabulary size V = 37286, total number of words in corpus ΣN = 3072662 (therefore average document size (ΣN)/M ≈ 70.5), and maximum document size max N = 307.

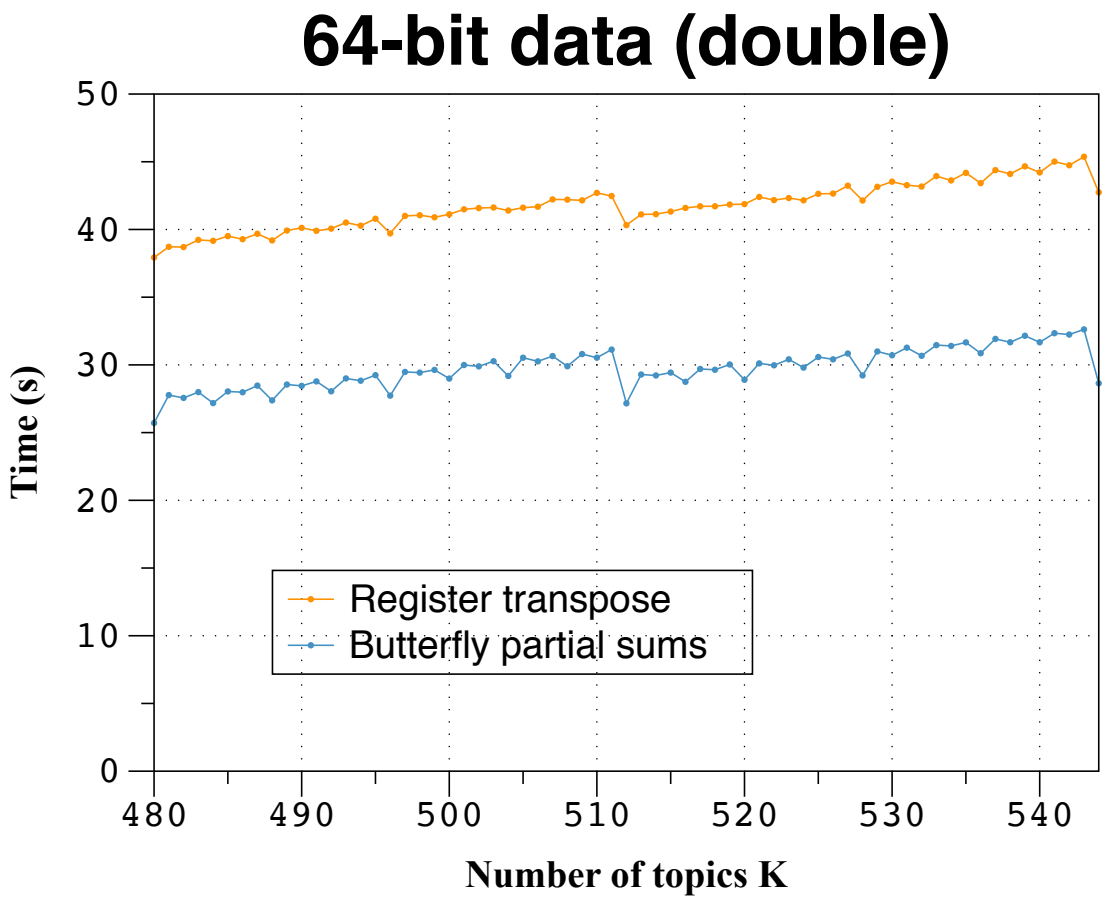
Measurements on Our Machine-Learning Application (2016)

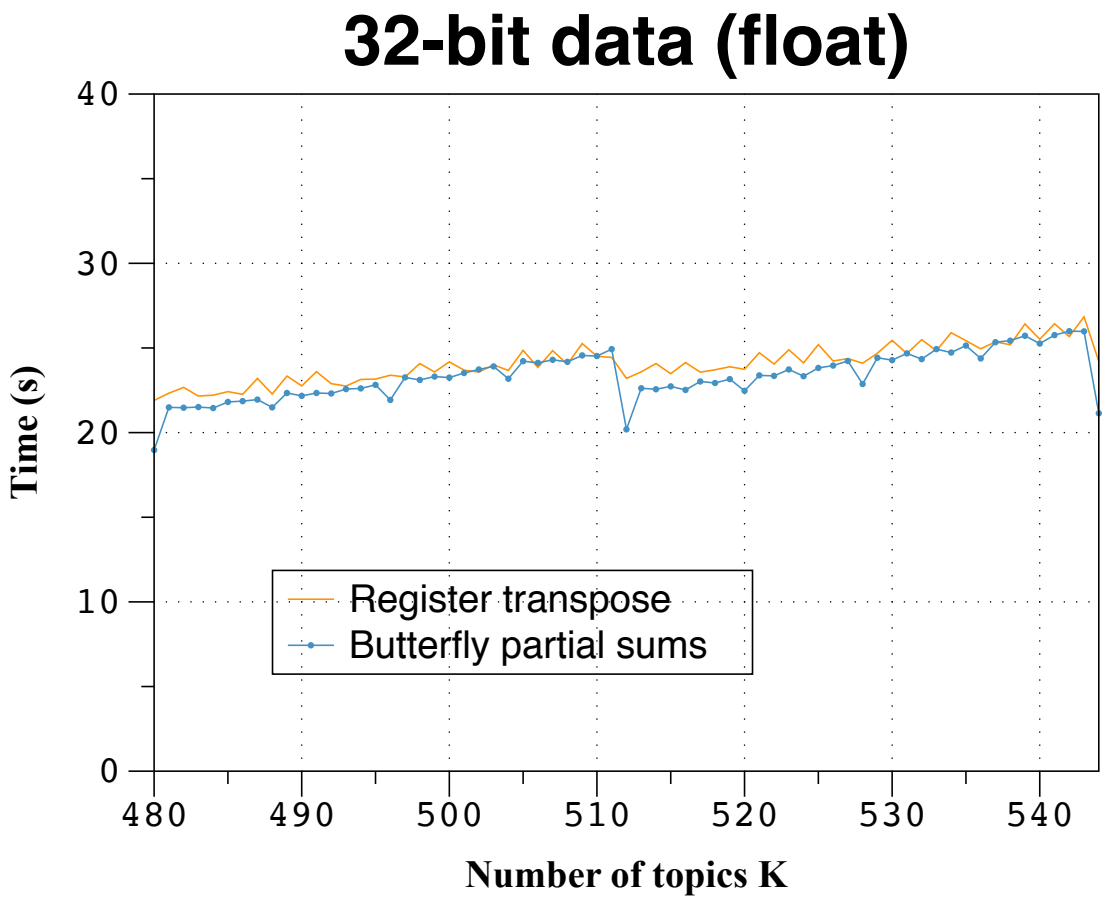




Coded in CUDA 7.5 for a single NVIDIA Titan Black GPU. Wikipedia-based dataset with number of documents M = 43556, vocabulary size V = 37286, total number of words in corpus ΣN = 3072662 (therefore average document size (ΣN)/M ≈ 70.5), and maximum document size max N = 307.

Measurements on Our Machine-Learning Application (2016)





Coded in CUDA 7.5 for a single NVIDIA Titan Black GPU. Wikipedia-based dataset with number of documents M = 43556, vocabulary size V = 37286, total number of words in corpus ΣN = 3072662 (therefore average document size (ΣN)/M ≈ 70.5), and maximum document size max N = 307.

Conclusions

- Transposed access to main-memory arrays can have a big payoff!
- Using transposed memory accesses, butterfly-patterned partial sums, and a modified binary search improved the overall speed of our machine-learning application by a factor of two.
- The butterfly pattern is up to 35% faster than transposed access alone.
- It pays to minimize the work to build the table, even at the expense of making the binary search much more complicated, because the binary search touches very few table entries (4 instead of 16, 5 instead of 32).
- We believe this technique may be applicable in other applications that need to draw from discrete distributions that are computed on the fly and used just once.