Understanding efficient parallel scan

Conal Elliott

October, 2013

Prefix sum (left scan)

$$\boxed{a_1, \ldots, a_n}$$

$$\mathit{lscan} \Downarrow$$

$$\boxed{b_1, \ldots, b_n}\boxed{b_{n+1}}$$

where $$b_k = \sum_{1 \leq i < k} a_i$$

In CUDA C

```
__global__ void prescan(float *g_odata, float *g_idata, int n) {
    extern __shared__ float temp[];  // allocated on invocation
    int thid = threadIdx.x;
    int offset = 1;
    // load input into shared memory
    temp[2*thid] = g_idata[2*thid];
    temp[2*thid+1] = g_idata[2*thid+1];
    // build sum in place up the tree
    for (int d = n>>1; d > 0; d >>= 1) {
        __syncthreads();
        if (thid < d) {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            temp[bi] += temp[ai]; }
        offset *= 2; }
    // clear the last element
    if (thid == 0) { temp[n - 1] = 0; }
    // traverse down tree & build scan
    for (int d = 1; d < n; d *= 2) {
        offset >>= 1;
        __syncthreads();
        if (thid < d) {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            float t = temp[ai];
            temp[ai] = temp[bi];
            temp[bi] += t; } }
    __syncthreads();
    // write results to device memory
    g_odata[2*thid] = temp[2*thid];
    g_odata[2*thid+1] = temp[2*thid+1]; }
```

Source: Harris, Sengupta, and Owens in *GPU Gems 3*, Chapter 39

In CUDA C

```
__global__ void prescan(float *g_odata, float *g_idata, int n) {
    extern __shared__ float temp[];  // allocated on invocation
    int thid = threadIdx.x;
    int offset = 1;
    // load input into shared memory
    temp[2*thid] = g_idata[2*thid];
    temp[2*thid+1] = g_idata[2*thid+1];
    // build sum in place up the tree
    for (int d = n>>1; d > 0; d >>= 1) {
        __syncthreads();
        if (thid < d) {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            temp[bi] += temp[ai]; }
        offset *= 2; }
    // clear the last element
    if (thid == 0) { temp[n - 1] = 0; }
    // traverse down tree & build scan
    for (int d = 1; d < n; d *= 2) {
        offset >>= 1;
        __syncthreads();
        if (thid < d) {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            float t = temp[ai];
            temp[ai] = temp[bi];
            temp[bi] += t; } }
    __syncthreads();
    // write results to device memory
    g_odata[2*thid] = temp[2*thid];
    g_odata[2*thid+1] = temp[2*thid+1]; }
```

WAT

Source: Harris, Sengupta, and Owens in *GPU Gems 3*, Chapter 39

In NESL

```
function scan(a) =
if #a == 1 then [0]
else
  let es = even_elts(a);
      os = odd_elts(a);
      ss = scan({e+o: e in es; o in os})
  in interleave(ss,{s+e: s in ss; e in es})
```

Source: Guy Blelloch in *Programming parallel algorithms, 1990*

In NESL

```
function scan(a) =
if #a == 1 then [0]
else
  let es = even_elts(a);
      os = odd_elts(a);
      ss = scan({e+o: e in es; o in os})
  in interleave(ss,{s+e: s in ss; e in es})
```

Source: Guy Blelloch in *Programming parallel algorithms, 1990*

Still, why does it work?

Prefix sum (left scan)

$$\boxed{a_1, \ldots, a_n} \;\; \mathit{lscan} \Downarrow \;\; \boxed{b_1, \ldots, b_n}\boxed{b_{n+1}}$$

where

$$b_k = \sum_{1 \le i < k} a_i$$

Prefix sum (left scan)

$$\boxed{a_1, \ldots, a_n} \quad \xrightarrow{lscan \Downarrow} \quad \boxed{b_1, \ldots, b_n}\,\boxed{b_{n+1}}$$

where

$$b_k = \sum_{1 \le i < k} a_i$$

Work: $O(n^2)$.

Prefix sum (left scan)

$$\boxed{a_1, \ldots, a_n} \; \overset{lscan \Downarrow}{} \; \boxed{b_1, \ldots, b_n}\boxed{b_{n+1}} \qquad \text{where} \qquad b_k = \sum_{1 \leq i < k} a_i$$

Work: $O(n^2)$.

Time:

Prefix sum (left scan)

$$\begin{array}{c} \boxed{a_1, \ldots, a_n} \\ \mathit{lscan} \Downarrow \\ \boxed{b_1, \ldots, b_n}\boxed{b_{n+1}} \end{array} \quad \text{where} \quad b_k = \sum_{1 \leq i < k} a_i$$

Work: $O(n^2)$.

Time: $O(n^2)$, $O(n)$, $O(\log n)$.

As a recurrence

$$\boxed{a_1, \ldots, a_n} \quad \text{where} \quad b_1 = 0$$

$$\mathit{lscan} \Downarrow \qquad b_{k+1} = b_k + a_k$$

$$\boxed{b_1, \ldots, b_n}\boxed{b_{n+1}}$$

As a recurrence

$$\boxed{a_1, \ldots, a_n} \quad \Big\Downarrow \; lscan \quad \boxed{b_1, \ldots, b_n}\boxed{b_{n+1}}$$

where

$$\begin{aligned} b_1 &= 0 \\ b_{k+1} &= b_k + a_k \end{aligned}$$

Work: $O(n)$.

As a recurrence

$$\boxed{a_1, \ldots, a_n} \quad \text{where} \quad b_1 = 0$$
$$\mathit{lscan} \Downarrow \qquad\qquad b_{k+1} = b_k + a_k$$
$$\boxed{b_1, \ldots, b_n}\boxed{b_{n+1}}$$

Work: $O(n)$.

Depth (ideal parallel "time"): $O(n)$.

As a recurrence

$$\boxed{a_1, \ldots, a_n} \quad \textit{lscan} \Downarrow \quad \boxed{b_1, \ldots, b_n}\boxed{b_{n+1}}$$

where

$$b_1 = 0$$
$$b_{k+1} = b_k + a_k$$

Work: $O(n)$.

Depth (ideal parallel "time"): $O(n)$.

Linear *dependency chain* thwarts parallelism (depth $<$ work).

Divide and conquer

Divide and conquer

$$\boxed{a_1, \ldots, a_n, a'_1, \ldots, a'_n}$$

Divide and conquer

$$\boxed{a_1, \ldots, a_n, a'_1, \ldots, a'_n}$$

$$\Downarrow_{split}$$

Divide and conquer

$$\boxed{a_1, \ldots, a_n, a'_1, \ldots, a'_n}$$

$$\Downarrow_{split}$$

$$\boxed{a_1, \ldots, a_n} \; \boxed{a'_1, \ldots, a'_n}$$

Divide and conquer

$$\boxed{a_1, \ldots, a_n, a'_1, \ldots, a'_n}$$

$$\Downarrow_{split}$$

$$\boxed{a_1, \ldots, a_n} \; \boxed{a'_1, \ldots, a'_n}$$

$$\Downarrow_{lscan} \qquad\qquad \Downarrow_{lscan}$$

Divide and conquer

$$\boxed{a_1, \ldots, a_n, a'_1, \ldots, a'_n}$$

$$\Downarrow_{split}$$

$$\boxed{a_1, \ldots, a_n} \; \boxed{a'_1, \ldots, a'_n}$$

$$\Downarrow_{lscan} \qquad \Downarrow_{lscan}$$

$$\boxed{b_1, \ldots, b_n}\boxed{b_{n+1}} \; \boxed{b'_1, \ldots, b'_n}\boxed{b'_{n+1}}$$

Divide and conquer

$$\boxed{a_1, \ldots, a_n, a'_1, \ldots, a'_n}$$

$$\Downarrow_{split}$$

$$\boxed{a_1, \ldots, a_n} \; \boxed{a'_1, \ldots, a'_n}$$

$$\Downarrow_{lscan} \qquad \Downarrow_{lscan}$$

$$\boxed{b_1, \ldots, b_n}\boxed{b_{n+1}} \; \boxed{b'_1, \ldots, b'_n}\boxed{b'_{n+1}}$$

$$\Downarrow_{merge}$$

Divide and conquer

$$\boxed{a_1, \ldots, a_n, a'_1, \ldots, a'_n}$$

$$\Downarrow_{split}$$

$$\boxed{a_1, \ldots, a_n} \; \boxed{a'_1, \ldots, a'_n}$$

$$\Downarrow_{lscan} \qquad \Downarrow_{lscan}$$

$$\boxed{b_1, \ldots, b_n}\boxed{b_{n+1}} \; \boxed{b'_1, \ldots, b'_n}\boxed{b'_{n+1}}$$

$$\Downarrow_{merge}$$

$$\boxed{b_1, \ldots, b_n, b_{n+1} + b'_1, \ldots, b_{n+1} + b'_n}\boxed{b_{n+1} + b'_{n+1}}$$

Divide and conquer

$$\boxed{a_1, \ldots, a_n, a'_1, \ldots, a'_n}$$

$$\Downarrow_{split}$$

$$\boxed{a_1, \ldots, a_n} \; \boxed{a'_1, \ldots, a'_n}$$

$$\Downarrow_{lscan} \qquad \Downarrow_{lscan}$$

$$\boxed{b_1, \ldots, b_n}\boxed{b_{n+1}} \; \boxed{b'_1, \ldots, b'_n}\boxed{b'_{n+1}}$$

$$\Downarrow_{merge}$$

$$\boxed{b_1, \ldots, b_n, b_{n+1} + b'_1, \ldots, b_{n+1} + b'_n}\boxed{b_{n+1} + b'_{n+1}}$$

- Equivalent? Why?

Divide and conquer

$$\boxed{a_1, \ldots, a_n, a'_1, \ldots, a'_n}$$

$$\Downarrow_{split}$$

$$\boxed{a_1, \ldots, a_n} \; \boxed{a'_1, \ldots, a'_n}$$

$$\Downarrow_{lscan} \qquad \Downarrow_{lscan}$$

$$\boxed{b_1, \ldots, b_n}\boxed{b_{n+1}} \; \boxed{b'_1, \ldots, b'_n}\boxed{b'_{n+1}}$$

$$\Downarrow_{merge}$$

$$\boxed{b_1, \ldots, b_n, b_{n+1} + b'_1, \ldots, b_{n+1} + b'_n}\boxed{b_{n+1} + b'_{n+1}}$$

- Equivalent? Why? (Associativity.)

Divide and conquer

$$\boxed{a_1, \ldots, a_n, a'_1, \ldots, a'_n}$$

$$\Downarrow_{split}$$

$$\boxed{a_1, \ldots, a_n} \; \boxed{a'_1, \ldots, a'_n}$$

$$\Downarrow_{lscan} \qquad \Downarrow_{lscan}$$

$$\boxed{b_1, \ldots, b_n}\boxed{b_{n+1}} \; \boxed{b'_1, \ldots, b'_n}\boxed{b'_{n+1}}$$

$$\Downarrow_{merge}$$

$$\boxed{b_1, \ldots, b_n, b_{n+1} + b'_1, \ldots, b_{n+1} + b'_n}\boxed{b_{n+1} + b'_{n+1}}$$

- Equivalent? Why? (Associativity.)
- No more linear dependency chain.

Divide and conquer

$$\boxed{a_1, \ldots, a_n, a'_1, \ldots, a'_n}$$

$$\Downarrow_{split}$$

$$\boxed{a_1, \ldots, a_n} \; \boxed{a'_1, \ldots, a'_n}$$

$$\Downarrow_{lscan} \qquad \Downarrow_{lscan}$$

$$\boxed{b_1, \ldots, b_n}\boxed{b_{n+1}} \; \boxed{b'_1, \ldots, b'_n}\boxed{b'_{n+1}}$$

$$\Downarrow_{merge}$$

$$\boxed{b_1, \ldots, b_n, b_{n+1} + b'_1, \ldots, b_{n+1} + b'_n}\boxed{b_{n+1} + b'_{n+1}}$$

- Equivalent? Why? (Associativity.)
- No more linear dependency chain.
- Work and depth analysis?

Depth analysis

Depends on depth of splitting and merging.

Depth analysis

Depends on depth of splitting and merging.

- Constant:

$$D(n) = D(n/2) + O(1)$$
$$D(n) = O(\log n)$$

Depth analysis

Depends on depth of splitting and merging.

- Constant:

$$D(n) = D(n/2) + O(1)$$
$$D(n) = O(\log n)$$

- Linear:

$$D(n) = D(n/2) + O(n)$$
$$D(2^k) = O(1 + 2 + 4 + \cdots + 2^k) = O(2^k)$$
$$D(n) = O(n)$$

Depth analysis

Depends on depth of splitting and merging.

- Constant:

$$D(n) = D(n/2) + O(1)$$
$$D(n) = O(\log n)$$

- Linear:

$$D(n) = D(n/2) + O(n)$$
$$D(2^k) = O(1 + 2 + 4 + \cdots + 2^k) = O(2^k)$$
$$D(n) = O(n)$$

- Logarithmic:

$$D(n) = D(n/2) + O(\log n)$$
$$D(2^k) = O(0 + 1 + 2 + \cdots + k) = O(k^2)$$
$$D(n) = O(\log^2 n)$$

Work analysis

Work recurrence:

$$W(n) = 2\,W(n/2) + O(n)$$

Work analysis

Work recurrence:

$$W(n) = 2\,W(n/2) + O(n)$$

By the *Master Theorem*,

$$W(n) = O(n \log n)$$

Analysis summary

Sequential:

$$D(n) = O(n)$$
$$W(n) = O(n)$$

Analysis summary

Sequential:

$$D(n) = O(n)$$
$$W(n) = O(n)$$

Divide and conquer:

$$D(n) = O(\log n)$$
$$W(n) = O(n \, \log n)$$

Analysis summary

Sequential:

$$D(n) = O(n)$$
$$W(n) = O(n)$$

Divide and conquer:

$$D(n) = O(\log n)$$
$$W(n) = O(n \, \log n)$$

Challenge: can we get $O(n)$ work and $O(\log n)$ depth?

Master Theorem

Given a recurrence:

$$f(n) = a\, f(n/b) + O(n^d)$$

We have the following closed form bound:

$$f(n) = \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log n) & \text{if } a = b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Master Theorem ($d = 1$)

Given a recurrence:

$$f(n) = a\,f(n/b) + O(n)$$

We have the following closed form bound:

$$f(n) = \begin{cases} O(n) & \text{if } a < b \\ O(n\ \log n) & \text{if } a = b \\ O(n^{\log_b a}) & \text{if } a > b \end{cases}$$

Master Theorem ($d = 1$)

Given a recurrence:

$$f(n) = a\,f(n/b) + O(n)$$

We have the following closed form bound:

$$f(n) = \begin{cases} O(n) & \text{if } a < b \\ O(n\ \log n) & \text{if } a = b \\ O(n^{\log_b a}) & \text{if } a > b \end{cases}$$

Puzzle: how to get $a < b$ for our recurrence?

$$W(n) = 2\,W(n/2) + O(n)$$

Return to this question later.

Variation: 3-way split/merge

$$\boxed{a_{1,1}, \ldots, a_{1,m}, a_{2,1}, \ldots, a_{2,m}, a_{3,1}, \ldots, a_{3,m}}$$

Variation: 3-way split/merge

$$\boxed{a_{1,1}, \ldots, a_{1,m}, a_{2,1}, \ldots, a_{2,m}, a_{3,1}, \ldots, a_{3,m}}$$

$$\Downarrow_{split}$$

$$\boxed{a_{1,1}, \ldots, a_{1,m}} \; \boxed{a_{2,1}, \ldots, a_{2,m}} \; \boxed{a_{3,1}, \ldots, a_{3,m}}$$

Variation: 3-way split/merge

$$\boxed{a_{1,1}, \ldots, a_{1,m}, a_{2,1}, \ldots, a_{2,m}, a_{3,1}, \ldots, a_{3,m}}$$

$$\Downarrow_{split}$$

$$\boxed{a_{1,1}, \ldots, a_{1,m}} \; \boxed{a_{2,1}, \ldots, a_{2,m}} \; \boxed{a_{3,1}, \ldots, a_{3,m}}$$

$$\Downarrow_{lscan} \qquad \Downarrow_{lscan} \qquad \Downarrow_{lscan}$$

$$\boxed{b_{1,1}, \ldots, b_{1,m}}\boxed{b_{1,m+1}} \; \boxed{b_{2,1}, \ldots, b_{2,m}}\boxed{b_{2,m+1}} \; \boxed{b_{3,1}, \ldots, b_{3,m}}\boxed{b_{3,m+1}}$$

Variation: 3-way split/merge

$$\boxed{a_{1,1},\ldots,a_{1,m},a_{2,1},\ldots,a_{2,m},a_{3,1},\ldots,a_{3,m}}$$

$$\Downarrow_{split}$$

$$\boxed{a_{1,1},\ldots,a_{1,m}}\ \boxed{a_{2,1},\ldots,a_{2,m}}\ \boxed{a_{3,1},\ldots,a_{3,m}}$$

$$\Downarrow_{lscan}\qquad\Downarrow_{lscan}\qquad\Downarrow_{lscan}$$

$$\boxed{b_{1,1},\ldots,b_{1,m}}\boxed{b_{1,m+1}}\ \boxed{b_{2,1},\ldots,b_{2,m}}\boxed{b_{2,m+1}}\ \boxed{b_{3,1},\ldots,b_{3,m}}\boxed{b_{3,m+1}}$$

$$\Downarrow_{merge}$$

$$\boxed{d_{1,1},\ldots,d_{1,m},d_{2,1},\ldots,d_{2,m},d_{3,1},\ldots,d_{3,m}}\boxed{d_{3,m+1}}$$

where

$$\begin{aligned} d_{1,j} &= b_{1,j} \\ d_{2,j} &= b_{1,m+1} + b_{2,j} \\ d_{3,j} &= b_{1,m+1} + b_{2,m+1} + b_{3,j} \end{aligned}$$

Variation: k-way split/merge

$$\boxed{a_{1,1}, \ldots, a_{1,m}, \ldots, a_{k,1}, \ldots, a_{k,m}}$$

Variation: k-way split/merge

$$\boxed{a_{1,1}, \ldots, a_{1,m}, \ldots, a_{k,1}, \ldots, a_{k,m}}$$

$$\Downarrow_{split}$$

$$\boxed{a_{1,1}, \ldots, a_{1,m}} \quad \ldots \quad \boxed{a_{k,1}, \ldots, a_{k,m}}$$

Variation: k-way split/merge

$$\boxed{a_{1,1}, \ldots, a_{1,m}, \ldots, a_{k,1}, \ldots, a_{k,m}}$$

$$\Downarrow_{split}$$

$$\boxed{a_{1,1}, \ldots, a_{1,m}} \quad \cdots \quad \boxed{a_{k,1}, \ldots, a_{k,m}}$$

$$\Downarrow_{lscan} \quad \cdots \quad \Downarrow_{lscan}$$

$$\boxed{b_{1,1}, \ldots, b_{1,m}}\boxed{b_{1,m+1}} \quad \cdots \quad \boxed{b_{k,1}, \ldots, b_{k,m}}\boxed{b_{k,m+1}}$$

Variation: k-way split/merge

$$\boxed{a_{1,1}, \ldots, a_{1,m}, \ldots, a_{k,1}, \ldots, a_{k,m}}$$

$$\Downarrow_{split}$$

$$\boxed{a_{1,1}, \ldots, a_{1,m}} \quad \cdots \quad \boxed{a_{k,1}, \ldots, a_{k,m}}$$

$$\Downarrow_{lscan} \quad \cdots \quad \Downarrow_{lscan}$$

$$\boxed{b_{1,1}, \ldots, b_{1,m}}\boxed{b_{1,m+1}} \quad \cdots \quad \boxed{b_{k,1}, \ldots, b_{k,m}}\boxed{b_{k,m+1}}$$

$$\Downarrow_{merge}$$

$$\boxed{d_{1,1}, \ldots, d_{1,m}, \ldots, d_{k,1}, \ldots, d_{k,m}}\boxed{c_{k+1}}$$

where

$$d_{i,j} = c_i + b_{i,j}$$

$$c_i = \sum_{1 \leq l < i} b_{l,m+1}$$

k-way split/merge

$$\boxed{a_{1,1}, \ldots, a_{1,m}, \ldots, a_{k,1}, \ldots, a_{k,m}}$$

$$\Downarrow_{split}$$

$$\boxed{a_{1,1}, \ldots, a_{1,m}} \quad \cdots \quad \boxed{a_{k,1}, \ldots, a_{k,m}}$$

$$\Downarrow_{lscan} \quad \cdots \quad \Downarrow_{lscan}$$

$$\boxed{b_{1,1}, \ldots, b_{1,m}}\boxed{b_{1,m+1}} \quad \cdots \quad \boxed{b_{k,1}, \ldots, b_{k,m}}\boxed{b_{k,m+1}}$$

$$\Downarrow_{merge}$$

$$\boxed{d_{1,1}, \ldots, d_{1,m}, \ldots, d_{k,1}, \ldots, d_{k,m}}\boxed{c_{k+1}}$$

where

$$d_{i,j} = c_j + b_{i,j}$$

$$\boxed{c_1, \ldots, c_k}\boxed{c_{k+1}} = lscan\left(\boxed{b_{1,m+1}, \ldots, b_{k,m+1}}\right)$$

k-way split/merge

$$\boxed{a_{1,1}, \ldots, a_{1,m}, \ldots, a_{k,1}, \ldots, a_{k,m}}$$

$$\Downarrow_{split}$$

$$\boxed{a_{1,1}, \ldots, a_{1,m}} \quad \cdots \quad \boxed{a_{k,1}, \ldots, a_{k,m}}$$

$$\Downarrow_{lscan} \quad \cdots \quad \Downarrow_{lscan}$$

$$\boxed{b_{1,1}, \ldots, b_{1,m}}\boxed{b_{1,m+1}} \quad \cdots \quad \boxed{b_{k,1}, \ldots, b_{k,m}}\boxed{b_{k,m+1}}$$

$$\Downarrow_{merge}$$

$$\boxed{d_{1,1}, \ldots, d_{1,m}, \ldots, d_{k,1}, \ldots, d_{k,m}}\boxed{c_{k+1}}$$

where

$$\boxed{b_{1,m+1}, \ldots, b_{k,m+1}}$$

$$\Downarrow_{lscan}$$

$$\boxed{c_1, \ldots, c_k}\boxed{c_{k+1}}$$

$$d_{i,j} = c_j + b_{i,j}$$

Work analysis

Master Theorem ($d = 1$):

$$W(n) = a\,W(n/b) + O(n)$$

$$W(n) = \begin{cases} O(n) & \text{if } a < b \\ O(n\ \log n) & \text{if } a = b \\ O(n^{\log_b a}) & \text{if } a > b \end{cases}$$

Work analysis

Master Theorem ($d = 1$):

$$W(n) = a\,W(n/b) + O(n)$$

$$W(n) = \begin{cases} O(n) & \text{if } a < b \\ O(n\ \log n) & \text{if } a = b \\ O(n^{\log_b a}) & \text{if } a > b \end{cases}$$

Scan with k-way split:

$$\begin{aligned} W(n) &= k\,W(n/k) + W(k) + O(n) \\ &= k\,W(n/k) + O(n) \end{aligned}$$

Still $O(n\ \log n)$.

Work analysis

Master Theorem ($d = 1$):

$$W(n) = a\,W(n/b) + O(n)$$

$$W(n) = \begin{cases} O(n) & \text{if } a < b \\ O(n\,\log n) & \text{if } a = b \\ O(n^{\log_b a}) & \text{if } a > b \end{cases}$$

Scan with k-way split:

$$\begin{aligned} W(n) &= k\,W(n/k) + W(k) + O(n) \\ &= k\,W(n/k) + O(n) \end{aligned}$$

Still $O(n\,\log n)$.

If k is *fixed*.

Split inversion

Two kinds of split:

- *Top-down* — k pieces of size n/k each

$$\begin{aligned} W(n) &= k\,W(n/k) + W(k) + O(n) \\ &= k\,W(n/k) + O(n) \\ &= O(n \log n) \end{aligned}$$

Split inversion

Two kinds of split:

- *Top-down* — k pieces of size n/k each

$$
\begin{aligned}
W(n) &= k\,W(n/k) + W(k) + O(n) \\
&= k\,W(n/k) + O(n) \\
&= O(n \log n)
\end{aligned}
$$

- *Bottom-up* — n/k pieces of size k each:

Split inversion

Two kinds of split:

- *Top-down* — k pieces of size n/k each

$$\begin{aligned} W(n) &= k\,W(n/k) + W(k) + O(n) \\ &= k\,W(n/k) + O(n) \\ &= O(n \log n) \end{aligned}$$

- *Bottom-up* — n/k pieces of size k each:

$$\begin{aligned} W(n) &= (n/k)\,W(k) + W(n/k) + O(n) \\ &= W(n/k) + O(n) \\ &= O(n) \end{aligned}$$

Split inversion

Two kinds of split:

- *Top-down* — k pieces of size n/k each

$$
\begin{aligned}
W(n) &= k\,W(n/k) + W(k) + O(n) \\
&= k\,W(n/k) + O(n) \\
&= O(n \log n)
\end{aligned}
$$

- *Bottom-up* — n/k pieces of size k each:

$$
\begin{aligned}
W(n) &= (n/k)\,W(k) + W(n/k) + O(n) \\
&= W(n/k) + O(n) \\
&= O(n)
\end{aligned}
$$

Mission accomplished: $O(n)$ work and $O(\log n)$ depth!

Root split

Another idea: split into $\sqrt{n}$ pieces of size $\sqrt{n}$ each.

Root split

Another idea: split into $\sqrt{n}$ pieces of size $\sqrt{n}$ each.

$$\begin{aligned} W(n) &= \sqrt{n} \cdot W(\sqrt{n}) + W(\sqrt{n}) + O(n) \\ &= \sqrt{n} \cdot W(\sqrt{n}) + O(n) \\ &= O(n \ \log \log n) \end{aligned}$$

$$D(n) = O(\log \log n)$$

Nearly constant depth and nearly linear work. Useful in practice?

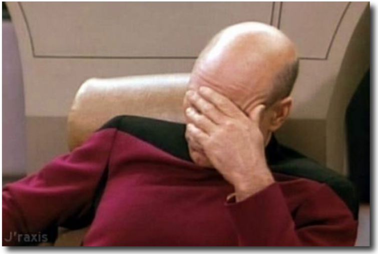
In CUDA C – bottom-up binary

```
__global__ void prescan(float *g_odata, float *g_idata, int n) {
    extern __shared__ float temp[];  // allocated on invocation
    int thid = threadIdx.x;
    int offset = 1;
    // load input into shared memory
    temp[2*thid] = g_idata[2*thid];
    temp[2*thid+1] = g_idata[2*thid+1];
    // build sum in place up the tree
    for (int d = n>>1; d > 0; d >>= 1) {
        __syncthreads();
        if (thid < d) {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            temp[bi] += temp[ai]; }
        offset *= 2; }
    // clear the last element
    if (thid == 0) { temp[n - 1] = 0; }
    // traverse down tree & build scan
    for (int d = 1; d < n; d *= 2) {
        offset >>= 1;
        __syncthreads();
        if (thid < d) {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            float t = temp[ai];
            temp[ai] = temp[bi];
            temp[bi] += t; } }
    __syncthreads();
    // write results to device memory
    g_odata[2*thid] = temp[2*thid];
    g_odata[2*thid+1] = temp[2*thid+1]; }
```

Source: Harris, Sengupta, and Owens in *GPU Gems 3*, Chapter 39

In CUDA C – bottom-up binary

```
__global__ void prescan(float *g_odata, float *g_idata, int n) {
    extern __shared__ float temp[];  // allocated on invocation
    int thid = threadIdx.x;
    int offset = 1;
    // load input into shared memory
    temp[2*thid] = g_idata[2*thid];
    temp[2*thid+1] = g_idata[2*thid+1];
    // build sum in place up the tree
    for (int d = n>>1; d > 0; d >>= 1) {
        __syncthreads();
        if (thid < d) {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            temp[bi] += temp[ai]; }
        offset *= 2; }
    // clear the last element
    if (thid == 0) { temp[n - 1] = 0; }
    // traverse down tree & build scan
    for (int d = 1; d < n; d *= 2) {
        offset >>= 1;
        __syncthreads();
        if (thid < d) {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            float t = temp[ai];
            temp[ai] = temp[bi];
            temp[bi] += t; } }
    __syncthreads();
    // write results to device memory
    g_odata[2*thid] = temp[2*thid];
    g_odata[2*thid+1] = temp[2*thid+1]; }
```



Source: Harris, Sengupta, and Owens in *GPU Gems 3*, Chapter 39

In Haskell — generalized left scan

$$\textbf{class}\ LScan\ c\ \textbf{where}$$
$$\quad lscan :: Monoid\ a \Rightarrow c\ a \rightarrow (c\ a, a)$$

Parametrized over container and associative operation.

Binary trees in Haskell

$\textbf{data}\ T\ a = L\ a \mid B\ (T\ a)\ (T\ a)$

Binary trees in Haskell

$$\textbf{data}\ T\ a = L\ a \mid B\ (T\ a)\ (T\ a)$$

Alternatively,

$$\textbf{data}\ T\ a = L\ a \mid B\ (Pair\ (T\ a))$$

where

$$\textbf{data}\ Pair\ a = a :\#\ a$$

Binary trees in Haskell

$$\textbf{data}\ T\ a = L\ a \mid B\ (T\ a)\ (T\ a)$$

Alternatively,

$$\textbf{data}\ T\ a = L\ a \mid B\ (Pair\ (T\ a))$$

where

$$\textbf{data}\ Pair\ a = a :\#\ a$$

Generalize from pairs:

$$\textbf{data}\ T\ f\ a = L\ a \mid B\ (f\ (T\ a))$$

In Haskell — top-down

$$\textbf{data}\ T\ f\ a = L\ a \mid B\ (f\ (T\ f\ a))$$

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Zippy}\ f, \mathit{LScan}\ f) \Rightarrow \mathit{LScan}\ (T\ f)\ \textbf{where} \\
\quad \mathit{lscan}\ (L\ a) \ \ = (L\ \emptyset, a) \\
\quad \mathit{lscan}\ (B\ \mathit{ts}) = (B\ (\mathit{adjust} <\$> \mathit{zip}\ (\mathit{tots}', \mathit{ts}')), \mathit{tot}) \\
\qquad \textbf{where} \\
\qquad\quad (\mathit{ts}', \mathit{tots}) \quad\ = \mathit{unzip}\ (\mathit{lscan} <\$> \mathit{ts}) \\
\qquad\quad (\mathit{tots}', \mathit{tot}) \quad = \mathit{lscan}\ \mathit{tots} \\
\qquad\quad \mathit{adjust}\ (p, t) = (p \oplus) <\$> t
\end{array}
$$

In Haskell — bottom-up

$$\textbf{data}\ T\ f\ a = L\ a \mid B\ (T\ f\ (f\ a))$$

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Zippy}\ f, \mathit{LScan}\ f) \Rightarrow \mathit{LScan}\ (T\ f)\ \textbf{where} \\
\quad \mathit{lscan}\ (L\ a)\ \ = (L\ \emptyset, a) \\
\quad \mathit{lscan}\ (B\ ts) = (B\ (\mathit{adjust} \mathbin{\langle\$\rangle} \mathit{zip}\ (\mathit{tots}', ts')), \mathit{tot}) \\
\qquad \textbf{where} \\
\qquad\quad (ts', \mathit{tots}) \quad\ = \mathit{unzip}\ (\mathit{lscan} \mathbin{\langle\$\rangle} ts) \\
\qquad\quad (\mathit{tots}', \mathit{tot}) \quad = \mathit{lscan}\ \mathit{tots} \\
\qquad\quad \mathit{adjust}\ (p, t) = (p\oplus) \mathbin{\langle\$\rangle} t
\end{array}
$$

In Haskell — root split

$$\textbf{data}\ T\ f\ a = L\ (f\ a) \mid B\ (T\ f\ (T\ f\ a))$$

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Zippy}\ f, \mathit{LScan}\ f) \Rightarrow \mathit{LScan}\ (T\ f)\ \textbf{where} \\
\quad \mathit{lscan}\ (L\ \mathit{as}) = \mathit{first}\ L\ (\mathit{lscan}\ \mathit{as}) \\
\quad \mathit{lscan}\ (B\ \mathit{ts}) = (B\ (\mathit{adjust} \mathbin{\ll\!\$\!\gg} \mathit{zip}\ (\mathit{tots}', \mathit{ts}')), \mathit{tot}) \\
\qquad \textbf{where} \\
\qquad\quad (\mathit{ts}', \mathit{tots}) \quad\ = \mathit{unzip}\ (\mathit{lscan} \mathbin{\ll\!\$\!\gg} \mathit{ts}) \\
\qquad\quad (\mathit{tots}', \mathit{tot}) \quad = \mathit{lscan}\ \mathit{tots} \\
\qquad\quad \mathit{adjust}\ (p, t) = (p \oplus) \mathbin{\ll\!\$\!\gg} t
\end{array}
$$

Type composition

$$\textbf{data}\ (g \circ f)\ a = \mathit{Comp}\ (g\ (f\ \ a))$$

Type composition

$$\textbf{data}\ (g \circ f)\ a = \mathit{Comp}\ (g\ (f\ \ a))$$

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Zippy}\ g, \mathit{LScan}\ g, \mathit{LScan}\ f) \Rightarrow \mathit{LScan}\ (g \circ f)\ \textbf{where} \\
\quad \mathit{lscan}\ (\mathit{Comp}\ \mathit{gfa}) = (\mathit{Comp}\ (\mathit{adjust} \mathbin{\langle\$\rangle} \mathit{zip}\ (\mathit{tots}', \mathit{gfa}')), \mathit{tot}) \\
\qquad \textbf{where} \\
\qquad\quad (\mathit{gfa}', \mathit{tots}) \quad\ \ = \mathit{unzip}\ (\mathit{lscan} \mathbin{\langle\$\rangle} \mathit{gfa}) \\
\qquad\quad (\mathit{tots}', \mathit{tot}) \quad\ \ = \mathit{lscan}\ \mathit{tots} \\
\qquad\quad \mathit{adjust}\ (p, \mathit{fa}') = (p \oplus) \mathbin{\langle\$\rangle} \mathit{fa}'
\end{array}
$$

Trees with explicit composition

$$\textbf{data}\ T\ f\ a = L\ a \mid B\ ((f \circ T\ f)\ a) \quad \text{-- top-down f-tree}$$

$$\textbf{data}\ T\ f\ a = L\ a \mid B\ ((T\ f \circ f)\ a) \quad \text{-- bottom-up f-tree}$$

$$\textbf{data}\ T\ f\ a = L\ (f\ a) \mid B\ ((T\ f \circ T\ f)\ a) \quad \text{-- top-down root f-tree}$$

$$\textbf{data}\ T\ f\ a = L\ (f\ a) \mid B\ (T\ (f \circ f)\ a) \quad \text{-- bottom-up root f-tree}$$

Trees with explicit composition

$$\textbf{data}\ T\ f\ a = L\ a \mid B\ ((f \circ T\ f)\ a) \quad \text{-- top-down f-tree}$$

$$\textbf{data}\ T\ f\ a = L\ a \mid B\ ((T\ f \circ f)\ a) \quad \text{-- bottom-up f-tree}$$

$$\textbf{data}\ T\ f\ a = L\ (f\ a) \mid B\ ((T\ f \circ T\ f)\ a) \quad \text{-- top-down root f-tree}$$

$$\textbf{data}\ T\ f\ a = L\ (f\ a) \mid B\ (T\ (f \circ f)\ a) \quad \text{-- bottom-up root f-tree}$$

f-trees:

$$\begin{aligned}
&\textbf{instance}\ (\mathit{Zippy}\ f, \mathit{LScan}\ f) \Rightarrow \mathit{LScan}\ (T\ f)\ \textbf{where} \\
&\quad \mathit{lscan}\ (L\ a) \ = (L\ \emptyset, a) \\
&\quad \mathit{lscan}\ (B\ w) = \mathit{first}\ B\ (\mathit{lscan}\ w)
\end{aligned}$$

Trees with explicit composition

$$\textbf{data}\ T\ f\ a = L\ a \quad \mid B\ ((f \circ T\ f)\quad a) \quad \text{-- top-down f-tree}$$

$$\textbf{data}\ T\ f\ a = L\ a \quad \mid B\ ((T\ f \circ f)\quad a) \quad \text{-- bottom-up f-tree}$$

$$\textbf{data}\ T\ f\ a = L\ (f\ a) \mid B\ ((T\ f \circ T\ f)\ a) \quad \text{-- top-down root f-tree}$$

$$\textbf{data}\ T\ f\ a = L\ (f\ a) \mid B\ (T\ (f \circ f)\quad a) \quad \text{-- bottom-up root f-tree}$$

Root f-trees:

$$\begin{array}{l} \textbf{instance}\ (\mathit{Zippy}\ f, \mathit{LScan}\ f) \Rightarrow \mathit{LScan}\ (T\ f)\ \textbf{where} \\ \quad \mathit{lscan}\ (L\ \mathit{as}) = \mathit{first}\ L\ \ (\mathit{lscan}\ \mathit{as}) \\ \quad \mathit{lscan}\ (B\ w) = \mathit{first}\ B\ (\mathit{lscan}\ w) \end{array}$$

Trees with explicit composition

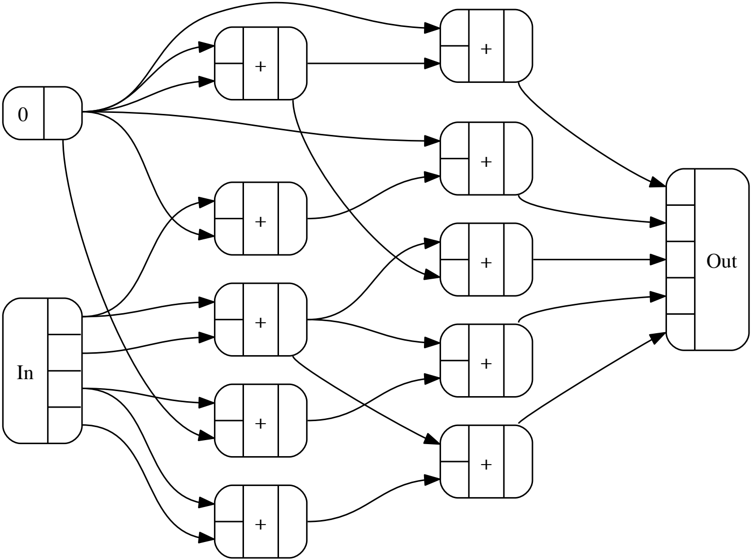
$$\begin{array}{llll}
\textbf{data}\ T\ f\ a = L\ a & \mid B\ ((f \circ T\ f) & a) & \text{-- top-down f-tree} \\
\textbf{data}\ T\ f\ a = L\ a & \mid B\ ((T\ f \circ f) & a) & \text{-- bottom-up f-tree} \\
\textbf{data}\ T\ f\ a = L\ (f\ a) & \mid B\ ((T\ f \circ T\ f) & a) & \text{-- top-down root f-tree} \\
\textbf{data}\ T\ f\ a = L\ (f\ a) & \mid B\ (T\ (f \circ f) & a) & \text{-- bottom-up root f-tree}
\end{array}$$

Root f-trees:

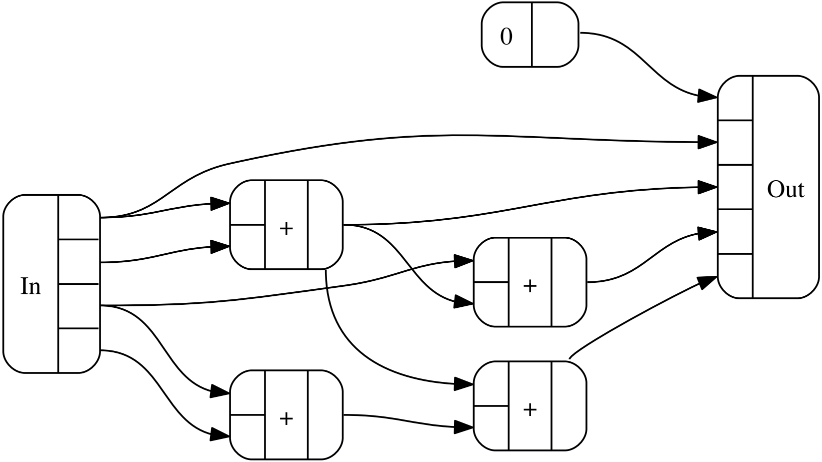
$$\begin{array}{l}
\textbf{instance}\ (\mathit{Zippy}\ f, \mathit{LScan}\ f) \Rightarrow \mathit{LScan}\ (T\ f)\ \textbf{where} \\
\quad \mathit{lscan}\ (L\ \mathit{as}) = \mathit{first}\ L\ (\mathit{lscan}\ \mathit{as}) \\
\quad \mathit{lscan}\ (B\ w) = \mathit{first}\ B\ (\mathit{lscan}\ w)
\end{array}$$

The bottom-up trees are *perfect* – f^n and f^{2^n}.

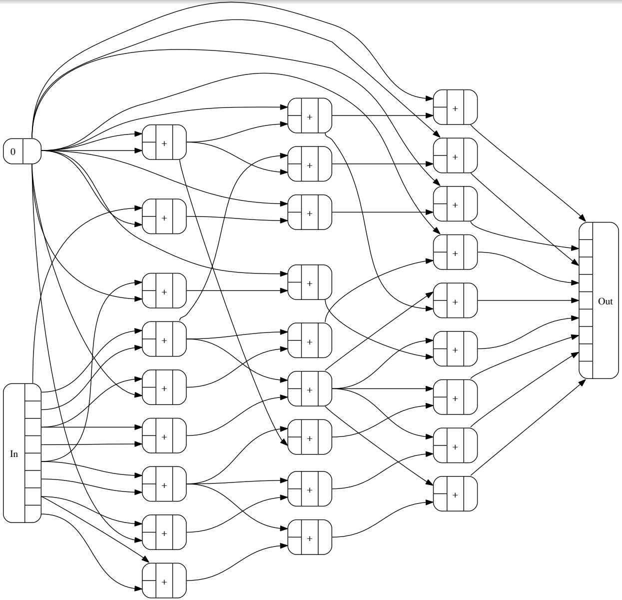
Top-down, depth 2



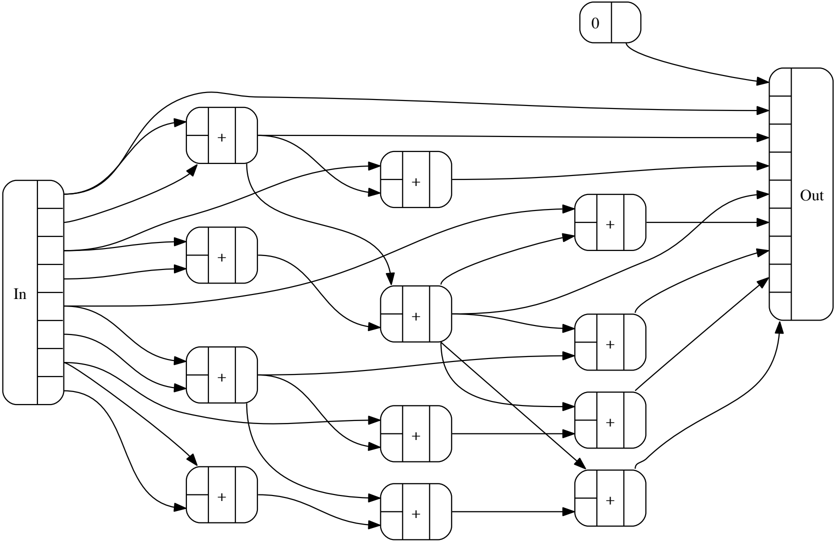
Top-down, depth 2, optimized



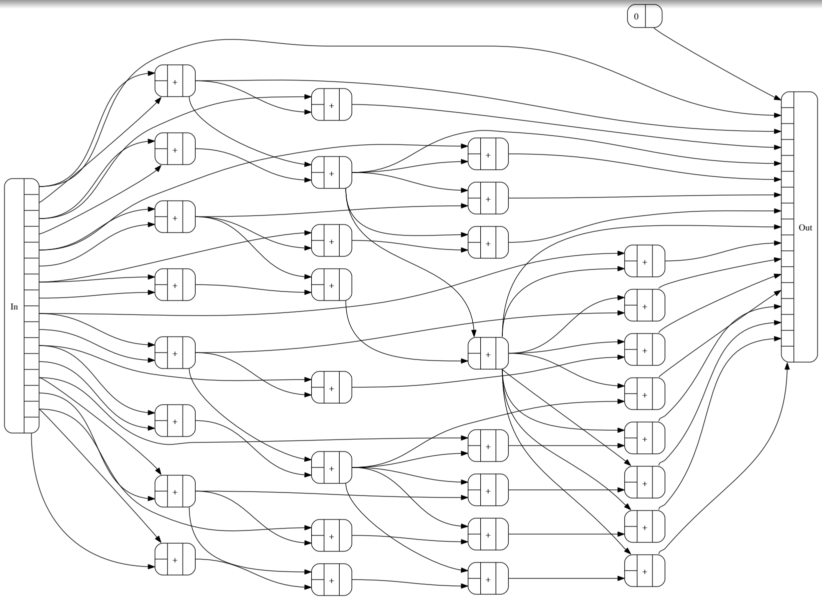
Top-down, depth 3



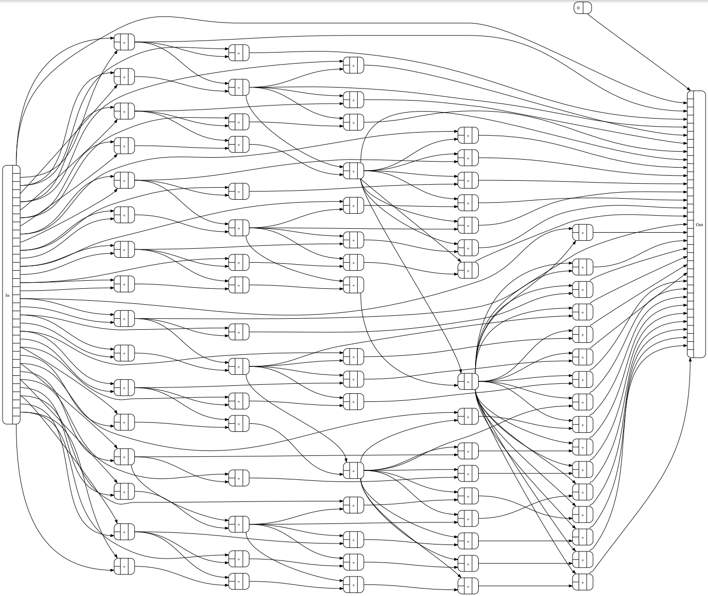
Top-down, depth 3, optimized



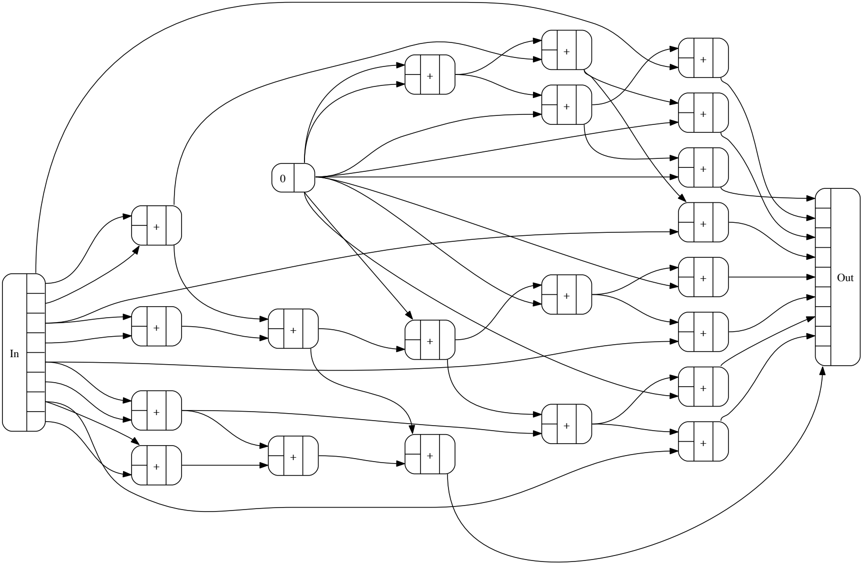
Top-down, depth 4, optimized



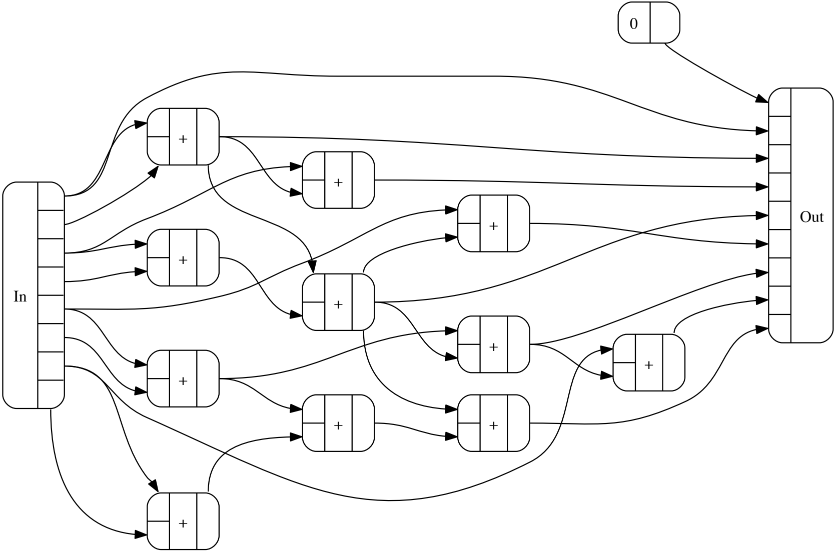
Top-down, depth 5, optimized



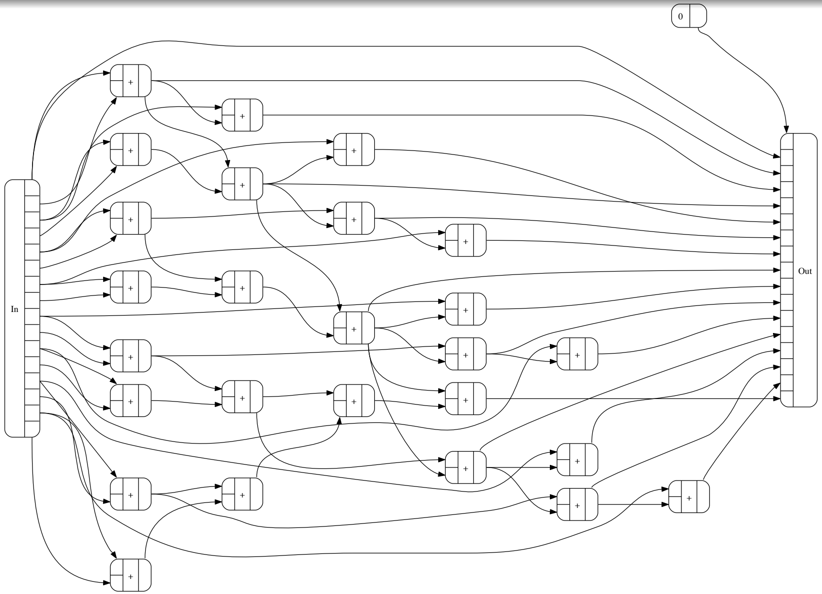
Bottom-up, depth 3



Bottom-up, depth 3, optimized



Bottom-up, depth 4, optimized



Data structure tinker toys

Data structure tinker toys

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Const}\ b & a = & \mathit{Const}\ b \\
\textbf{data}\ \mathit{Id} & a = & \mathit{Id}\ a \\
\textbf{data}\ (f \times g) & a = & \mathit{Prod}\ (f\ a)\ (g\ a) \\
\textbf{data}\ (f + g) & a = & \mathit{InL}\ (f\ a) \mid \mathit{InR}\ (g\ a) \\
\textbf{data}\ (g \circ f) & a = & \mathit{Comp}\ (g\ (f\ a))
\end{array}
$$

Data structure tinker toys

$$
\begin{aligned}
&\textbf{data}\ \mathit{Const}\ b\ a = \mathit{Const}\ b \\
&\textbf{data}\ \mathit{Id}\ \qquad a = \mathit{Id}\ a \\
&\textbf{data}\ (f \times g)\ a = \mathit{Prod}\ (f\ a)\ (g\ a) \\
&\textbf{data}\ (f + g)\ a = \mathit{InL}\ (f\ a) \mid \mathit{InR}\ (g\ a) \\
&\textbf{data}\ (g \circ f)\ a = \mathit{Comp}\ (g\ (f\ a))
\end{aligned}
$$

Each has an *LScan* instance.

Parallel scan for many data structures.

See post: *Composable parallel scanning*.

Data structure tinker toys

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Const}\ b & a = & \mathit{Const}\ b \\
\textbf{data}\ \mathit{Id} & a = & \mathit{Id}\ a \\
\textbf{data}\ (f \times g) & a = & \mathit{Prod}\ (f\ a)\ (g\ a) \\
\textbf{data}\ (f + g) & a = & \mathit{InL}\ (f\ a) \mid \mathit{InR}\ (g\ a) \\
\textbf{data}\ (g \circ f) & a = & \mathit{Comp}\ (g\ (f\ a))
\end{array}
$$

Each has an *LScan* instance.

Parallel scan for many data structures.

See post: *Composable parallel scanning*.

Similar algorithm decompositions?