Functional programming and parallelism

Conal Elliott

May 2016

What makes a language good for parallelism?

. . .

What makes a language *bad* for parallelism?

- Sequential bias
  - Primitive: assignment (state change)
  - Composition: *sequential* execution
  - "Von Neumann" languages (Fortran, C, Java, Python, ...)
- *Over-linearizes* algorithms.
- Hard to isolate accidental sequentiality.

Can we fix sequential languages?

- Throw in parallel composition.

- Oops:
    - Nondeterminism
    - Deadlock
    - Intractable reasoning

Can we *un-break* sequential languages?

Perfection is achieved not when there is nothing left to add,
but when there is nothing left to take away.

Antoine de Saint-Exupéry

Applications perform zillions of simple computations.

- Compute all at once?
- Oops — dependencies.
- Minimize dependencies!

Dependencies

- Three sources:
  1. Problem
  2. Algorithm
  3. Language
- Goals: eliminate #3, and reduce #2.

Dependency in sequential languages

- Built into sequencing: $A\,;B$
- Semantics: B begins where A ends.
- Why sequence?

Idea: remove *all* state

- And, with it,
  - mutation (assignment),
  - sequencing,
  - statements.
- Expression dependencies are specific & explicit.
- Remainder can be parallel.
- Contrast: "$A\,;B$" vs "$A+B$" vs "$(A+B)\times C$".

Programming without state

- Programming is calculation/math:
  - Precise & tractable reasoning (algebra),
  - . . . including optimization/transformation.
- No loss of expressiveness!
- "Functional programming" (value-oriented)
- Like arithmetic on big values

Sequential sum

C:

```
int sum(int arr[], int n) {
    int acc = 0;
    for (int i=0; i<n; i++)
      acc += arr[i];
    return acc;
}
```

Haskell:

$$
\begin{aligned}
& \mathit{sum} = \mathit{sumAcc}\ 0 \\
& \quad \textbf{where} \\
& \qquad \mathit{sumAcc}\ \mathit{acc}\ [\,] \qquad\quad = \mathit{acc} \\
& \qquad \mathit{sumAcc}\ \mathit{acc}\ (a : \mathit{as}) = \mathit{sumAcc}\ (\mathit{acc} + a)\ \mathit{as}
\end{aligned}
$$

Refactoring

$$sum = foldl\ (+)\ 0$$

where

$$\begin{aligned} foldl\ op\ acc\ [\,] &= acc \\ foldl\ op\ acc\ (a : as) &= foldl\ op\ (acc \text{ `}op\text{` } a)\ as \end{aligned}$$

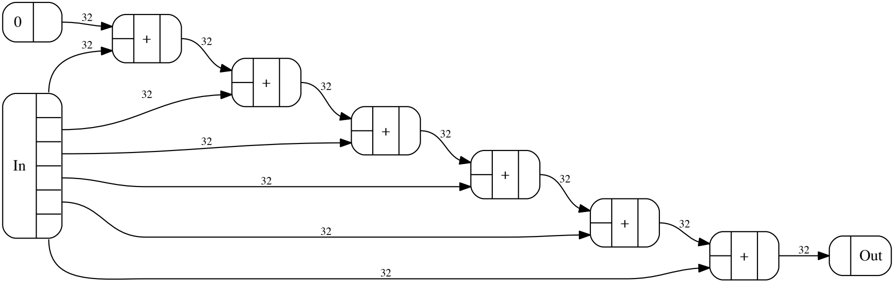
Right alternative

$$sum = foldr\ (+)\ 0$$

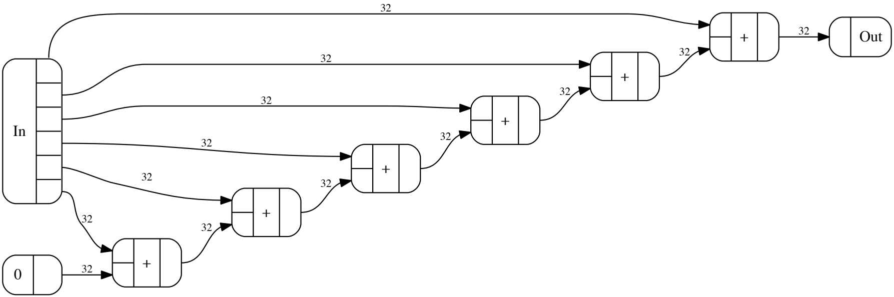
where

$$\begin{aligned} foldr\ op\ e\ [\,] &= e \\ foldr\ op\ e\ (a : as) &= a\ `op`\ foldr\ op\ e\ as \end{aligned}$$

Sequential sum — left



Sequential sum — right



Parallel sum — how?

Left-associated sum:

$$sum\ [a, b, ..., z] \equiv (...((0 + a) + b)...) + z$$

How to parallelize?

Divide and conquer?

Balanced data

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

Sequential:

$$\begin{aligned}
&\mathit{sum} = \mathit{sumAcc}\ 0\\
&\quad \textbf{where}\\
&\qquad \mathit{sumAcc}\ \mathit{acc}\ (L\ a) \quad\ \ = \mathit{acc} + a\\
&\qquad \mathit{sumAcc}\ \mathit{acc}\ (B\ s\ t) = \mathit{sumAcc}\ (\mathit{sumAcc}\ \mathit{acc}\ s)\ t
\end{aligned}$$

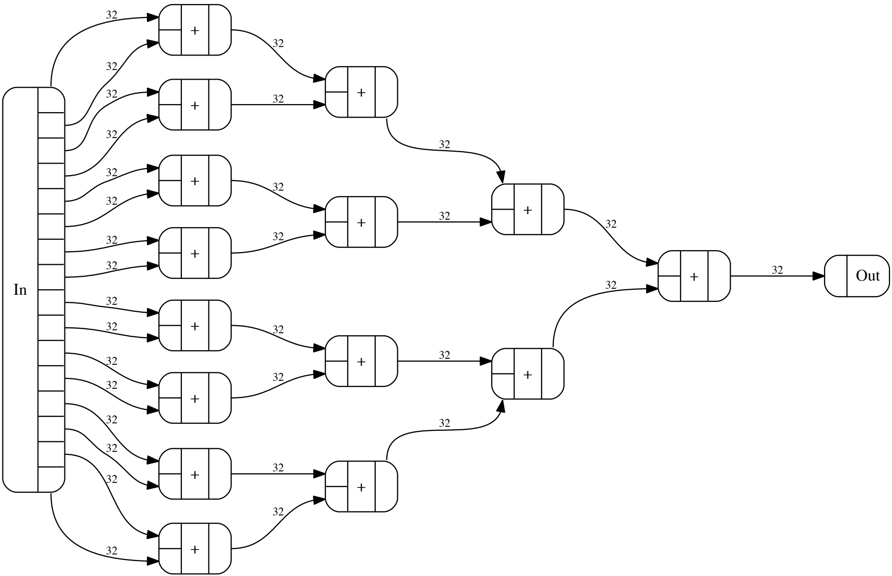
Again, $\mathit{sum} = \mathit{foldl}\ (+)\ 0$.

Parallel:

$$\begin{aligned}
&\mathit{sum}\ (L\ a) \quad\ \ = a\\
&\mathit{sum}\ (B\ s\ t) = \mathit{sum}\ s + \mathit{sum}\ t
\end{aligned}$$

Equivalent? Why?

Balanced tree sum — depth 4



Balanced computation

- Generalize beyond $+$, 0.
- When valid?

Associative folds

Monoid: type with associative operator & identity.

$$fold :: Monoid\ a \Rightarrow [\,a\,] \rightarrow a$$

Not just lists:

$$fold :: (Foldable\ f, Monoid\ a) \Rightarrow f\ a \rightarrow a$$

Balanced data structures lead to balanced parallelism.

Two associative folds

$$
\begin{aligned}
&\mathit{fold} :: \mathit{Monoid}\ a \Rightarrow [\,a\,] \to a \\
&\mathit{fold}\ [\,] \qquad\quad = \emptyset \\
&\mathit{fold}\ (a : \mathit{as}) = a \oplus \mathit{fold}\ \mathit{as}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{fold} :: \mathit{Monoid}\ a \Rightarrow \mathit{Tree}\ a \to a \\
&\mathit{fold}\ (L\ a) \quad\ = a \\
&\mathit{fold}\ (B\ s\ t) = \mathit{fold}\ s \oplus \mathit{fold}\ t
\end{aligned}
$$

Derivable automatically from types.

Trickier algorithm: prefix sums

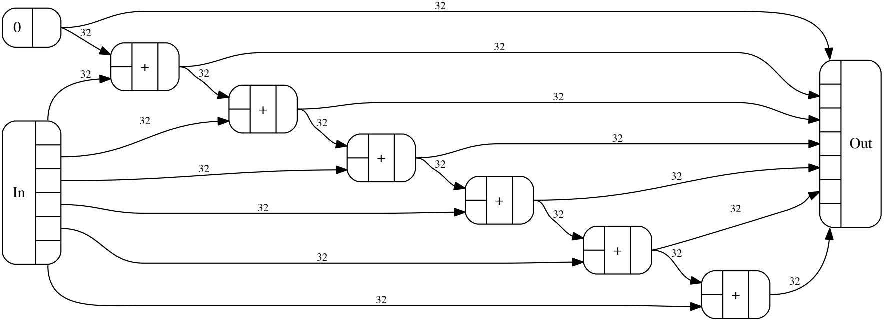
C:

```
int prefixSums(int arr[], int n) {
    int sum = 0;
    for (int i=0; i<n; i++) {
        int next = arr[i];
        arr[i] = sum;
        sum += next;
    }
    return sum;
}
```

Haskell:

$$\mathit{prefixSums} = \mathit{scanl}\ (+)\ 0$$

Sequence prefix sum

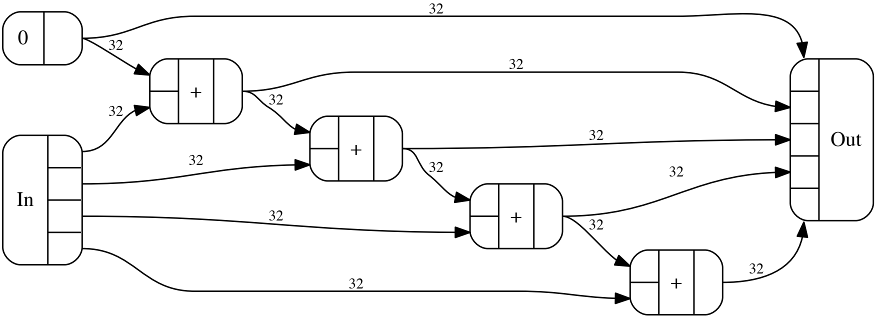


Sequential prefix sums on trees

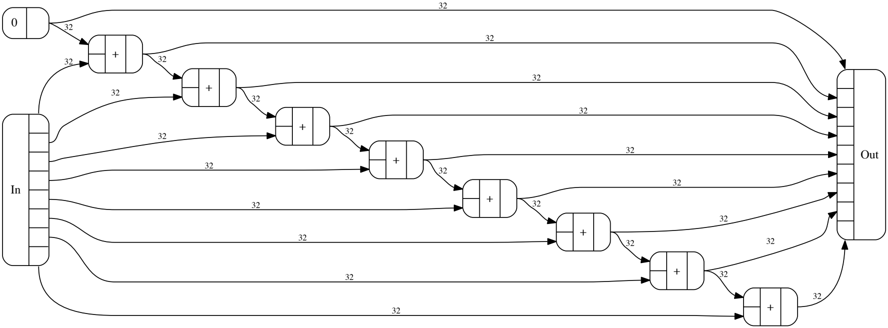
$$prefixSums = scanl\ (+)\ 0$$

$$\begin{aligned}
&scanl\ op\ acc\ (L\ a) \quad = (L\ acc, acc\ \text{`}op\text{`}\ a)\\
&scanl\ op\ acc\ (B\ u\ v) = (B\ u'\ v', vTot)\\
&\quad \mathbf{where}\\
&\qquad (u', uTot) = scanl\ op\ acc \quad u\\
&\qquad (v', vTot) = scanl\ op\ uTot\ v
\end{aligned}$$

Sequential prefix sums on trees — depth 2



Sequential prefix sums on trees — depth 3



Sequential prefix sums on trees

$$
\begin{array}{l}
\mathit{prefixSums} = \mathit{scanl}\ (+)\ 0 \\[1ex]
\mathit{scanl}\ \mathit{op}\ \mathit{acc}\ (L\ a) \quad\ = (L\ \mathit{acc}, \mathit{acc}\ `\mathit{op}`\ a) \\
\mathit{scanl}\ \mathit{op}\ \mathit{acc}\ (B\ u\ v) = (B\ u'\ v', \mathit{vTot}) \\
\quad \textbf{where} \\
\qquad (u', \mathit{uTot}) = \mathit{scanl}\ \mathit{op}\ \mathit{acc}\quad u \\
\qquad (v', \mathit{vTot}) = \mathit{scanl}\ \mathit{op}\ \mathit{uTot}\ v
\end{array}
$$

- Still very sequential.
- Does associativity help as with *fold*?

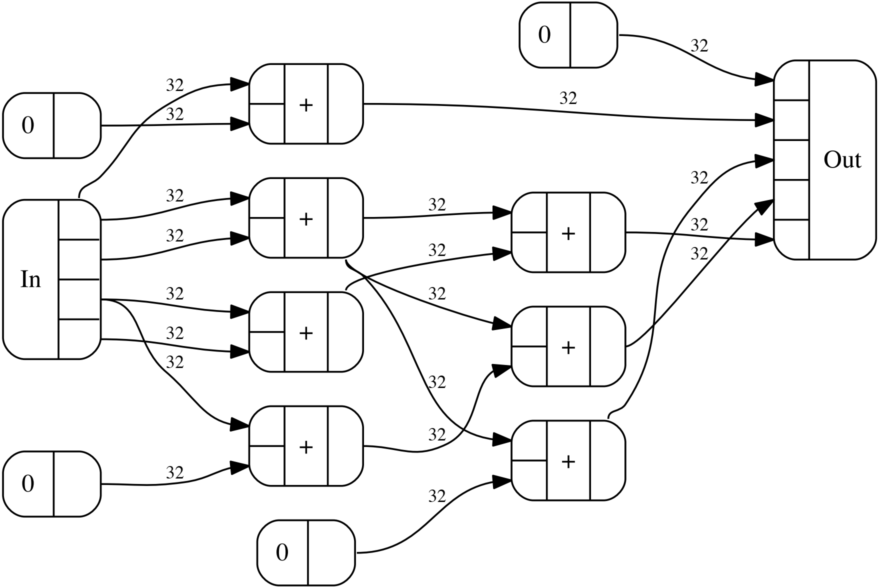
Parallel prefix sums on trees

On trees:

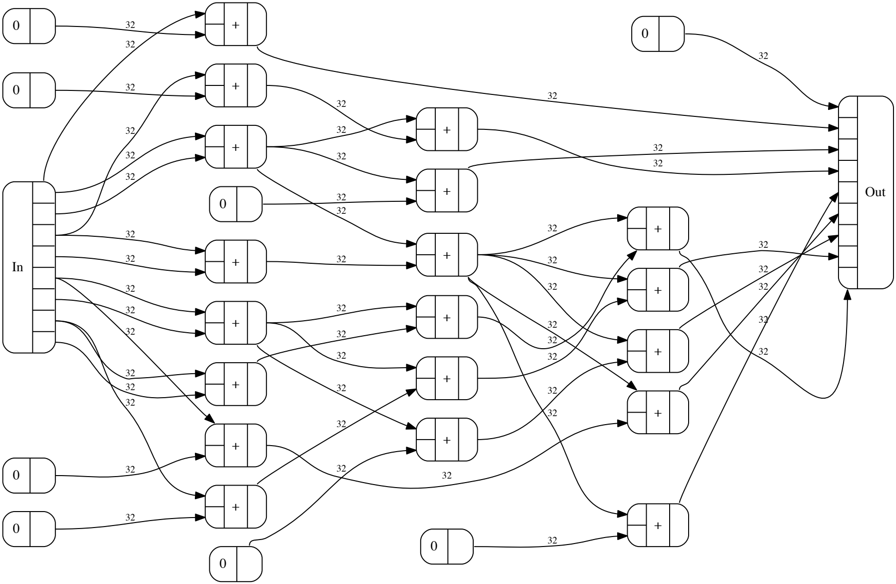
$$
\begin{aligned}
&\mathit{scan}\ (L\ a) &&= (L\ \emptyset, a) \\
&\mathit{scan}\ (B\ u\ v) &&= (B\ u'\ (\mathit{fmap}\ \mathit{adjust}\ v'), \mathit{adjust}\ \mathit{vTot}) \\
&\quad \textbf{where} \\
&\qquad (u', \mathit{uTot}) = \mathit{scan}\ u \\
&\qquad (v', \mathit{vTot}) = \mathit{scan}\ v \\
&\qquad \mathit{adjust}\ x \quad = \mathit{uTot} \oplus x
\end{aligned}
$$

- If balanced, dependency depth $O(\log n)$, work $O(n \log n)$.
- Can reduce work to $O(n)$. *(Understanding efficient parallel scan).*
- Generalizes from trees.
- *Automatic from type.*

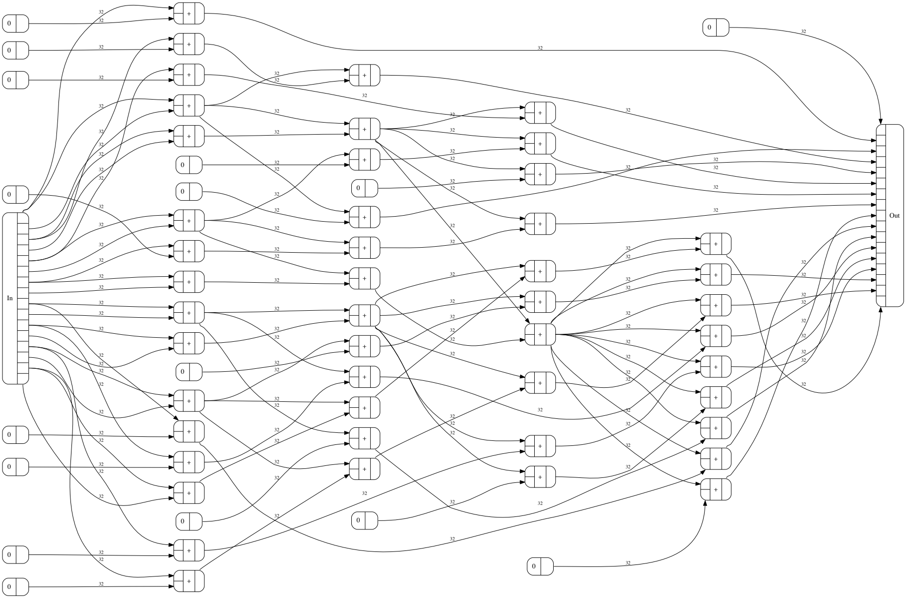
Balanced parallel prefix sums — depth 2



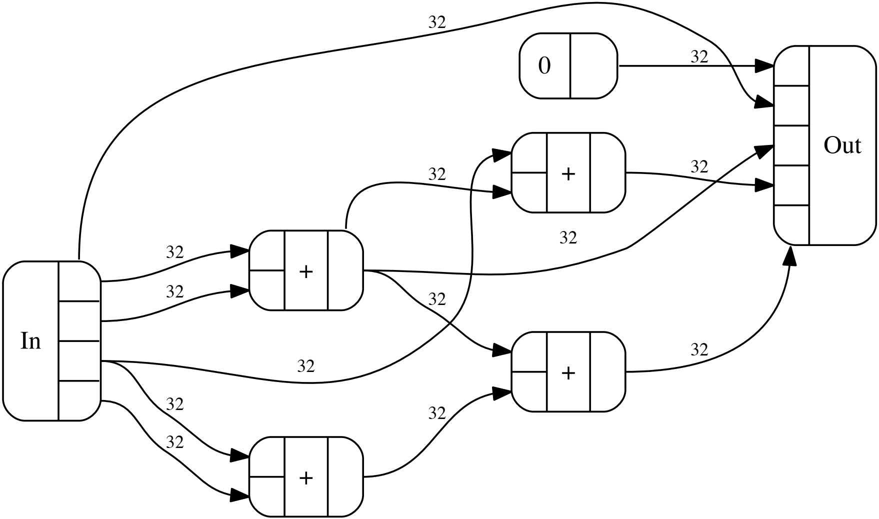
Balanced parallel prefix sums — depth 3



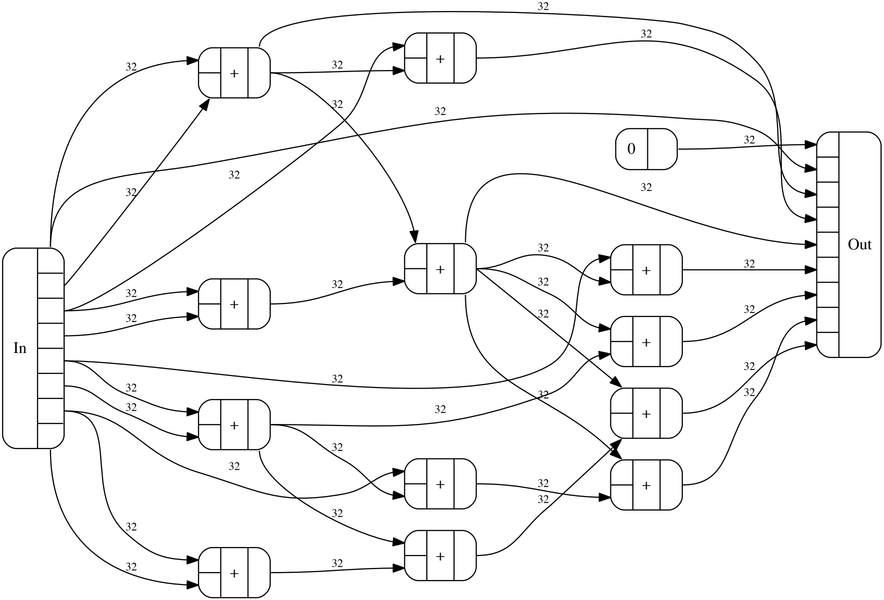
Balanced parallel prefix sums — depth 4



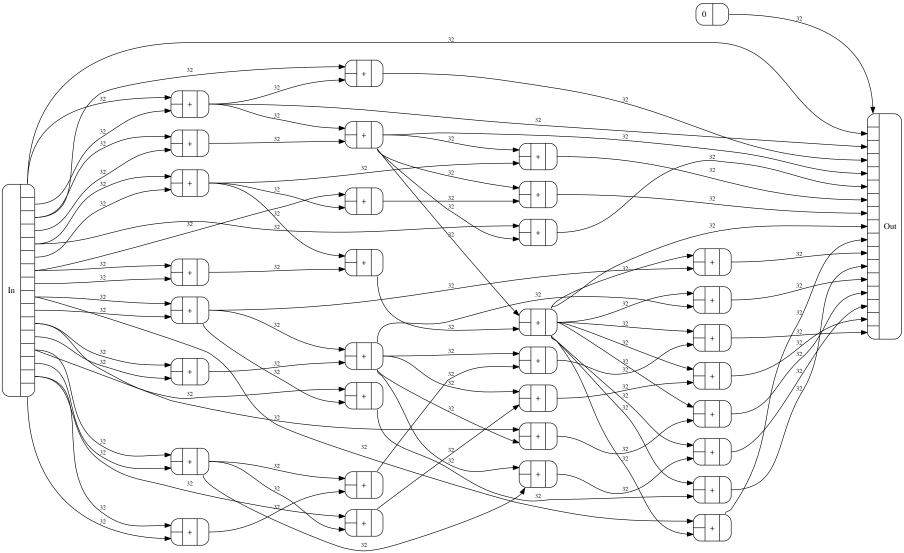
Balanced parallel prefix sums — depth 2, optimized



Balanced parallel prefix sums — depth 3, optimized



Balanced parallel prefix sums — depth 4, optimized



Why functional programming?

- Parallelism
- Correctness
- Productivity

R&D agenda: elegant, massively parallel FP

- Algorithm design:
  - Functional & richly typed
  - Parallel-friendly
  - Easily composable
- Compiling for highly parallel execution:
  - Convert to algebraic vocabulary (CCC).
  - Interpret vocabulary as "circuits" (FPGA, silicon, GPU).
  - Other interpretations.

Composable data structures

- Data structure tinker toys:

$$\begin{aligned}
&\mathbf{data}\ \mathit{Empty}\ \ a = \mathit{Empty} \\
&\mathbf{data}\ \mathit{Id}\qquad\ a = \mathit{Id}\ a \\
&\mathbf{data}\ (f + g)\ a = L\ (f\ \ a) \mid R\ (g\ \ a) \\
&\mathbf{data}\ (f \times g)\ a = \mathit{Prod}\ (f\ \ a)\ (g\ \ a) \\
&\mathbf{data}\ (g \circ f)\ a = O\ (g\ (f\ \ a))
\end{aligned}$$

- Specify algorithm version for each.
- Automatic, type-directed composition.

Vectors

$$\overbrace{Id \times \cdots \times Id}^{n \text{ times}}$$

Right-associated:

$$\begin{array}{l} \textbf{type family } \mathit{RVec}\ n \textbf{ where} \\ \quad \mathit{RVec}\ Z \quad\ \ = \mathit{Empty} \\ \quad \mathit{RVec}\ (S\ n) = \mathit{Id} \times \mathit{RVec}\ n \end{array}$$

Left-associated:

$$\begin{array}{l} \textbf{type family } \mathit{LVec}\ n \textbf{ where} \\ \quad \mathit{LVec}\ Z \quad\ \ = \mathit{Empty} \\ \quad \mathit{LVec}\ (S\ n) = \mathit{LVec}\ n \times \mathit{Id} \end{array}$$

Perfect binary leaf trees

$$\overbrace{Pair \circ \cdots \circ Pair}^{n \text{ times}}$$

Right-associated:

$$\begin{aligned}
&\textbf{type family } RBin\ n\ \textbf{where} \\
&\quad RBin\ Z \qquad\ = Id \\
&\quad RBin\ (S\ n) = Pair \circ RBin\ n
\end{aligned}$$

Left-associated:

$$\begin{aligned}
&\textbf{type family } LBin\ n\ \textbf{where} \\
&\quad LBin\ Z \qquad\ = Id \\
&\quad LBin\ (S\ n) = LBin\ n \circ Pair
\end{aligned}$$

Uniform pairs:

$$\textbf{type } Pair = Id \times Id$$

Generalized trees

$$\overbrace{h \circ \cdots \circ h}^{n \text{ times}}$$

Right-associated:

$$\begin{array}{l} \textbf{type family } \mathit{RPow}\ h\ n\ \textbf{where} \\ \quad \mathit{RPow}\ h\ Z \quad\ \ = \mathit{Id} \\ \quad \mathit{RPow}\ h\ (S\ n) = h \circ \mathit{RPow}\ h\ n \end{array}$$

Left-associated:

$$\begin{array}{l} \textbf{type family } \mathit{LPow}\ h\ n\ \textbf{where} \\ \quad \mathit{LPow}\ h\ Z \quad\ \ = \mathit{Id} \\ \quad \mathit{LPow}\ h\ (S\ n) = \mathit{LPow}\ h\ n \circ h \end{array}$$

Binary:

$$\begin{array}{l} \textbf{type } \mathit{RBin}\ n = \mathit{RPow}\ \mathit{Pair}\ n \\ \textbf{type } \mathit{LBin}\ n = \mathit{LPow}\ \mathit{Pair}\ n \end{array}$$

Composing scans

$$\textbf{class}\ \mathit{LScan}\ f\ \textbf{where}$$
$$\quad \mathit{lscan} :: \mathit{Monoid}\ a \Rightarrow f\ a \rightarrow (f \times \mathit{Id})\ a$$

$$\textbf{pattern}\ \mathit{And1}\ \mathit{fa}\ a = \mathit{Prod}\ \mathit{fa}\ (\mathit{Id}\ a)$$

$$\textbf{instance}\ \mathit{LScan}\ \mathit{Empty}\ \textbf{where}$$
$$\quad \mathit{lscan}\ \mathit{fa} = \mathit{And1}\ \mathit{fa}\ \emptyset$$

$$\textbf{instance}\ \mathit{LScan}\ \mathit{Id}\ \textbf{where}$$
$$\quad \mathit{lscan}\ (\mathit{Id}\ a) = \mathit{And1}\ (\mathit{Id}\ \emptyset)\ a$$

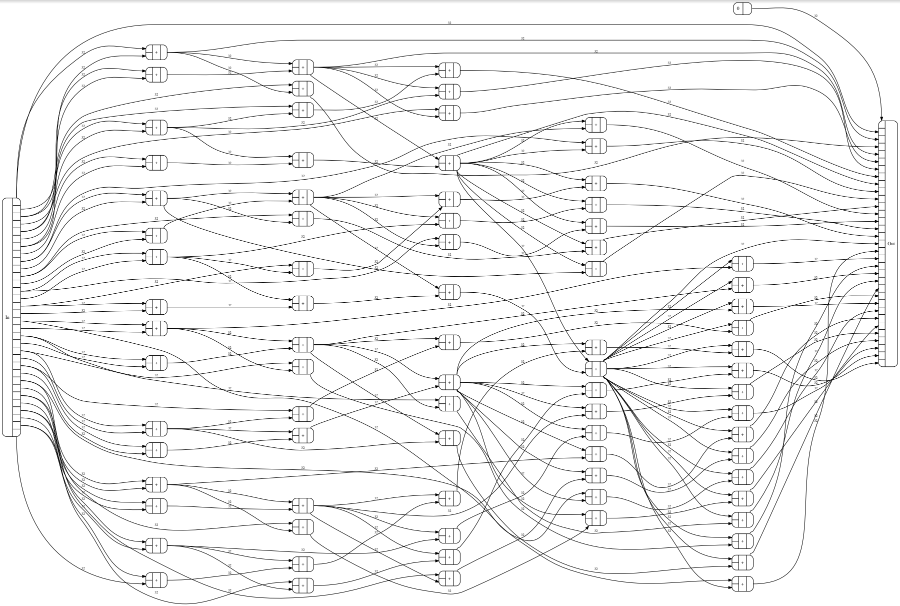
$$\textbf{instance}\ (\mathit{LScan}\ f, \mathit{LScan}\ g) \Rightarrow \mathit{LScan}\ (f \times g)\ \textbf{where}$$
$$\quad \mathit{lscan}\ (\mathit{Prod}\ \mathit{fa}\ \mathit{ga}) = \mathit{And1}\ (\mathit{Prod}\ \mathit{fa}'\ \mathit{ga}')\ \mathit{gx}$$
$$\quad\quad \textbf{where}$$
$$\quad\quad\quad \mathit{And1}\ \mathit{fa}'\ \mathit{fx} = \mathit{lscan}\ \mathit{fa}$$
$$\quad\quad\quad \mathit{And1}\ \mathit{ga}'\ \mathit{gx} = \mathit{adjust}\ \mathit{fx}\ (\mathit{lscan}\ \mathit{ga})$$

Composing scans

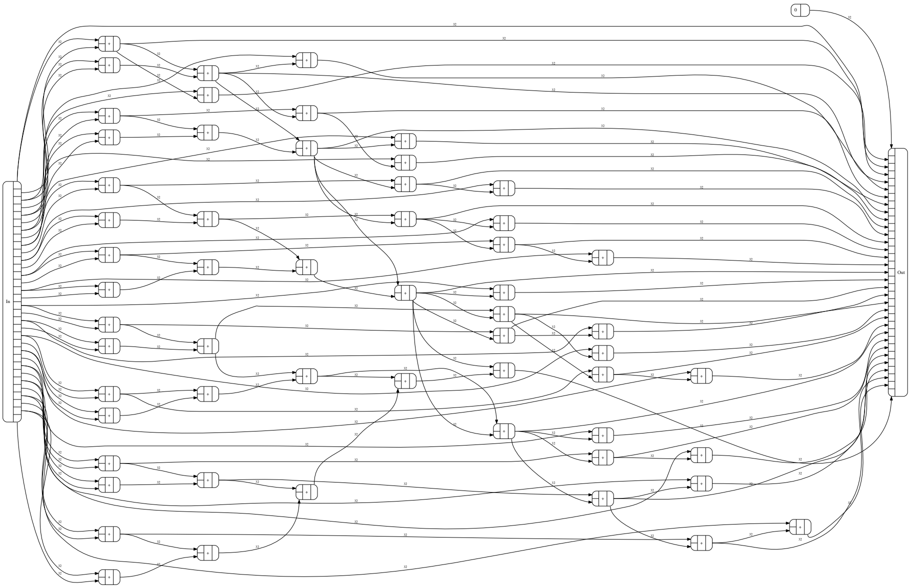
$$
\begin{aligned}
&\textbf{instance}\ (\mathit{LScan}\ g, \mathit{LScan}\ f, \mathit{Zip}\ g) \Rightarrow \mathit{LScan}\ (g \circ f)\ \textbf{where} \\
&\quad \mathit{lscan}\ (O\ \mathit{gfa}) = \mathit{And1}\ (O\ (\mathit{zipWith}\ \mathit{adjust}\ \mathit{tots}'\ \mathit{gfa}'))\ \mathit{tot} \\
&\qquad \textbf{where} \\
&\qquad\quad (\mathit{gfa}', \mathit{tots}) \qquad = \mathit{unzipAnd1}\ (\mathit{fmap}\ \mathit{lscan}\ \mathit{gfa}) \\
&\qquad\quad \mathit{And1}\ \mathit{tots}'\ \mathit{tot} = \mathit{lscan}\ \mathit{tots}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{adjust} :: (\mathit{Monoid}\ a, \mathit{Functor}\ t) \Rightarrow a \to t\ a \to t\ a \\
&\mathit{adjust}\ a\ t = \mathit{fmap}\ (a\oplus)\ t
\end{aligned}
$$

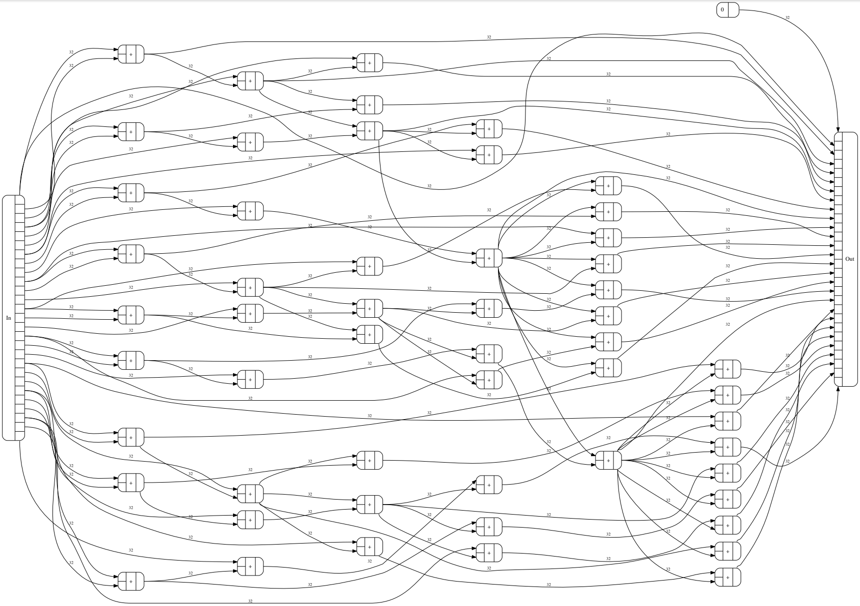
Scan — *RPow Pair N5*



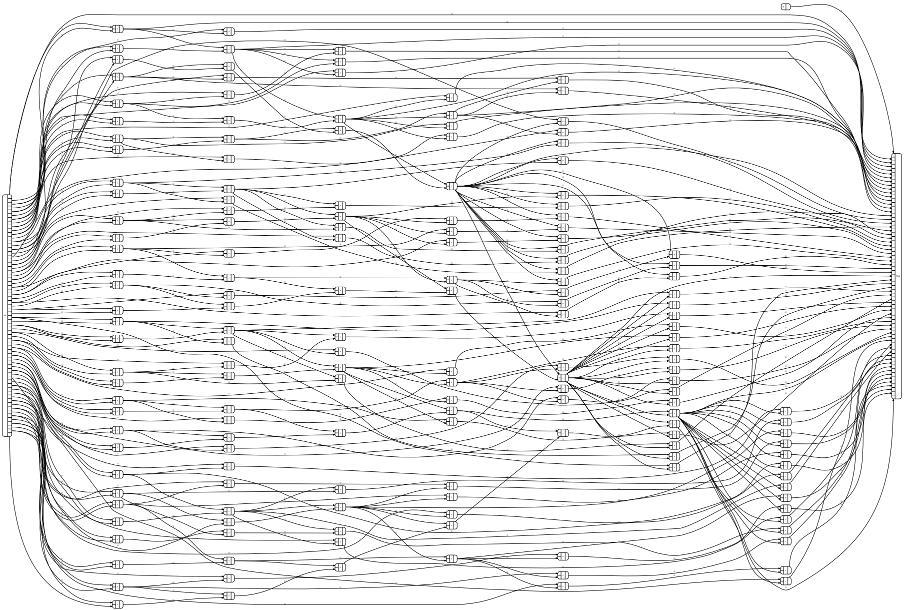
Scan — *LPow Pair N5*



Scan — *RPow (LVec N3) N3*



Scan — *RPow (LPow Pair N2) N3*



Polynomial evaluation

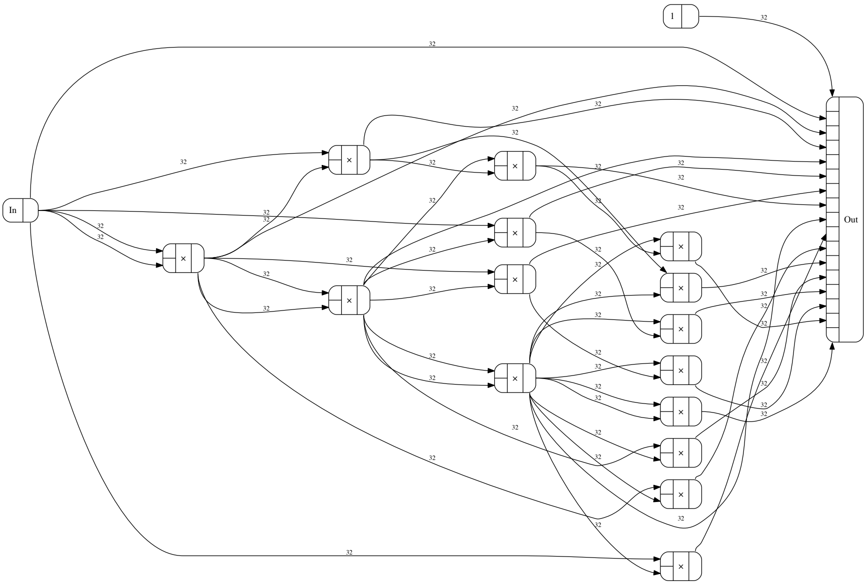
$$a_0 \cdot x^0 + \cdots + a_n \cdot x^n$$

$$\mathit{evalPoly}\ \mathit{coeffs}\ x = \mathit{coeffs} \cdot \mathit{powers}\ x$$

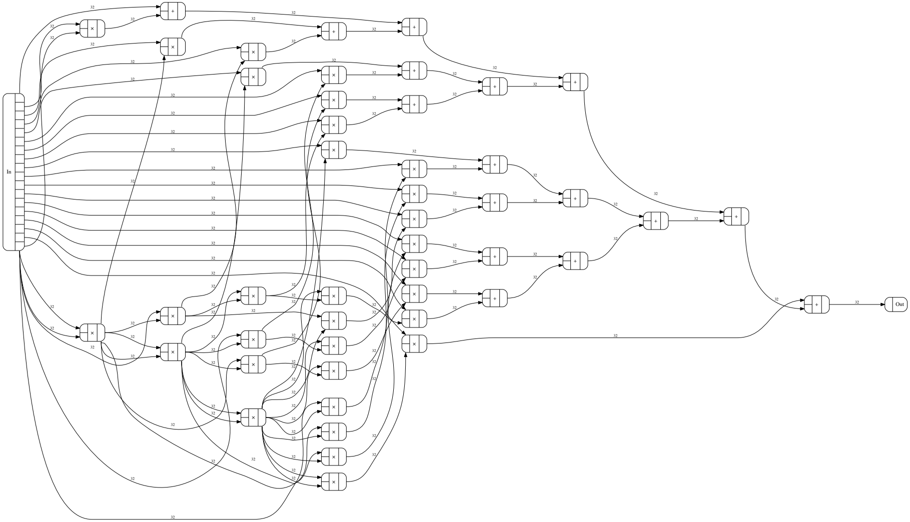
$$\mathit{powers} = \mathit{lproducts} \circ \mathit{pure}$$

$$\mathit{lproducts} = \mathit{underF}\ \mathit{Product}\ \mathit{lscan}$$

Powers — *RBin N4*



Polynomial evaluation — *RBin N4*



Fast Fourier transform

DFT:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk}$$

FFT for $N = N_1 \cdot N_2$ (Gauss / Cooley-Tukey):

$$X_k = \sum_{n_1=0}^{N_1-1} \left[e^{-\frac{2\pi i}{N} n_1 k_2} \right] \left(\sum_{n_2=0}^{N_2-1} x_{N_1 n_2 + n_1} e^{-\frac{2\pi i}{N_2} n_2 k_2} \right) e^{-\frac{2\pi i}{N_1} n_1 k_1}$$

Fast Fourier transform

$$
\begin{array}{l}
\textbf{class}\ \mathit{FFT}\ f\ \textbf{where} \\
\quad \textbf{type}\ \mathit{FFO}\ f :: * \to * \\
\quad \mathit{fft} :: \mathit{RealFloat}\ a \Rightarrow f\ (\mathit{Complex}\ a) \to \mathit{FFO}\ f\ (\mathit{Complex}\ a)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{FFT}\ \mathit{Id}\ \textbf{where} \\
\quad \textbf{type}\ \mathit{FFO}\ \mathit{Id} = \mathit{Id} \\
\quad \mathit{fft} = \mathit{id}
\end{array}
$$

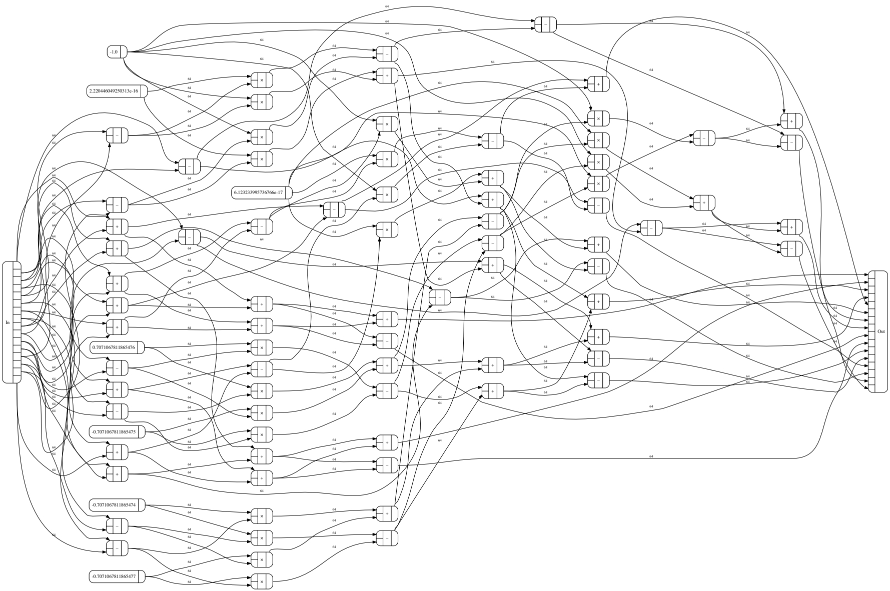
$$
\begin{array}{l}
\textbf{instance}\ \mathit{FFT}\ \mathit{Pair}\ \textbf{where} \\
\quad \textbf{type}\ \mathit{FFO}\ \mathit{Pair} = \mathit{Pair} \\
\quad \mathit{fft}\ (a :\!\#\ b) = (a + b) :\!\#\ (a - b)
\end{array}
$$

FFT — composition (Gauss / Cooley-Tukey)

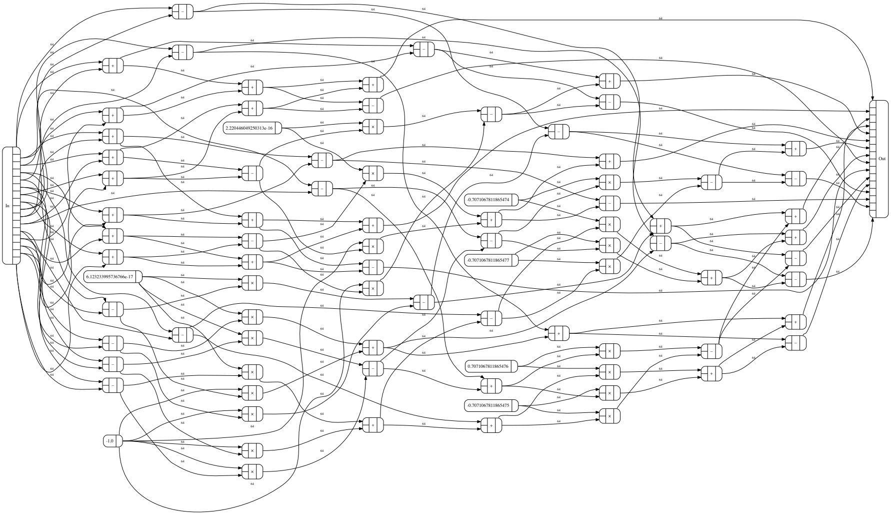
$$
\begin{array}{l}
\textbf{instance} \ldots \Rightarrow \mathit{FFT}\ (g \circ f)\ \textbf{where} \\
\quad \textbf{type}\ \mathit{FFO}\ (g \circ f) = \mathit{FFO}\ f \circ \mathit{FFO}\ g \\
\quad \mathit{fft} = O \circ \mathit{traverse}\ \mathit{fft} \circ \mathit{twiddle} \circ \mathit{traverse}\ \mathit{fft} \circ \mathit{transpose} \circ \mathit{unO}
\end{array}
$$

$$
\begin{array}{l}
\mathit{twiddle} :: \ldots \Rightarrow g\ (f\ (\mathit{Complex}\ a)) \to g\ (f\ (\mathit{Complex}\ a)) \\
\mathit{twiddle} = (\mathit{zipWith} \circ \mathit{zipWith})\ (*)\ \mathit{twiddles} \\
\quad \textbf{where} \\
\qquad \begin{array}{ll}
n & = \mathit{size}@(g \circ f) \\
\mathit{twiddles} & = \mathit{fmap}\ \mathit{powers}\ (\mathit{powers}\ \omega) \\
\omega & = \mathit{cis}\ (-2 * \pi\ /\ \mathit{fromIntegral}\ n) \\
\mathit{cis}\ a & = \mathit{cos}\ a :+ \mathit{sin}\ a
\end{array}
\end{array}
$$

FFT — *RBin N3* (“Decimation in time”)

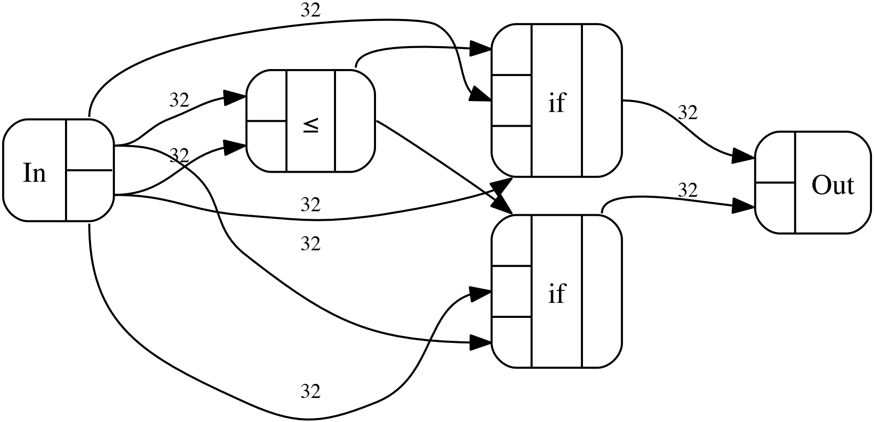


FFT — *LBin N3* (“Decimation in frequency”)

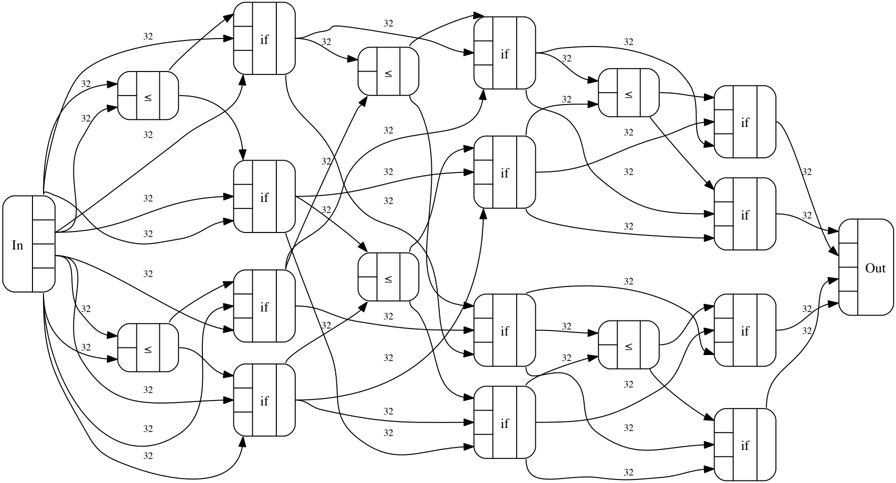


Bitonic sort

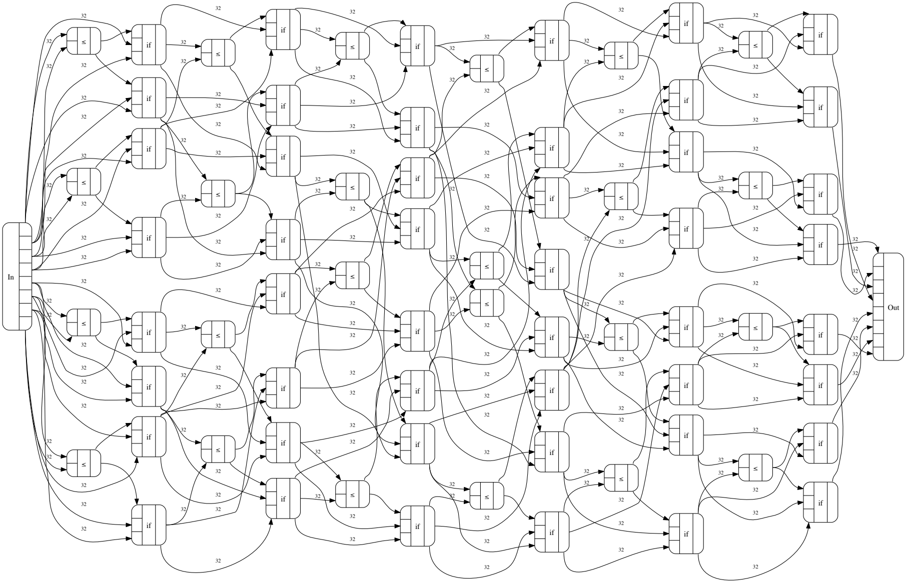
Bitonic sort — depth 1



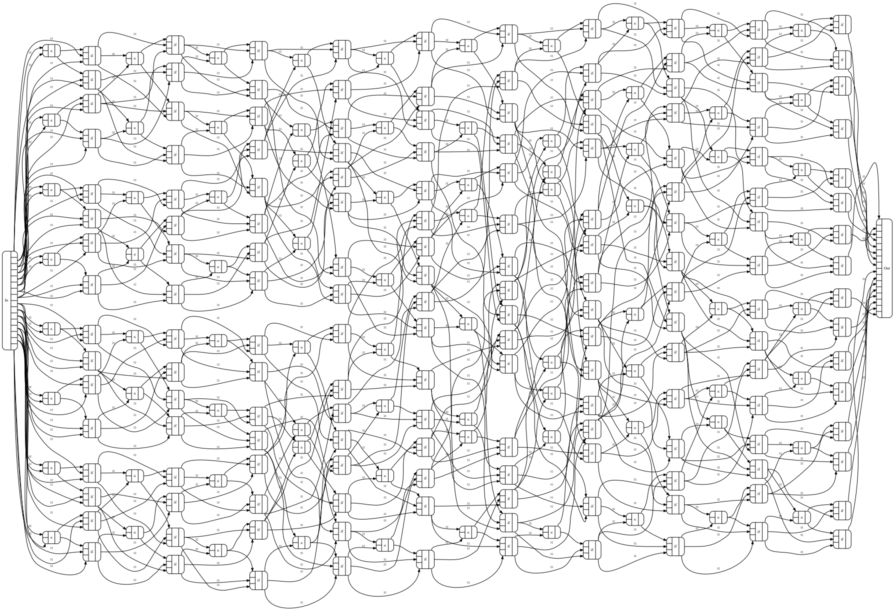
Bitonic sort — depth 2



Bitonic sort — depth 3

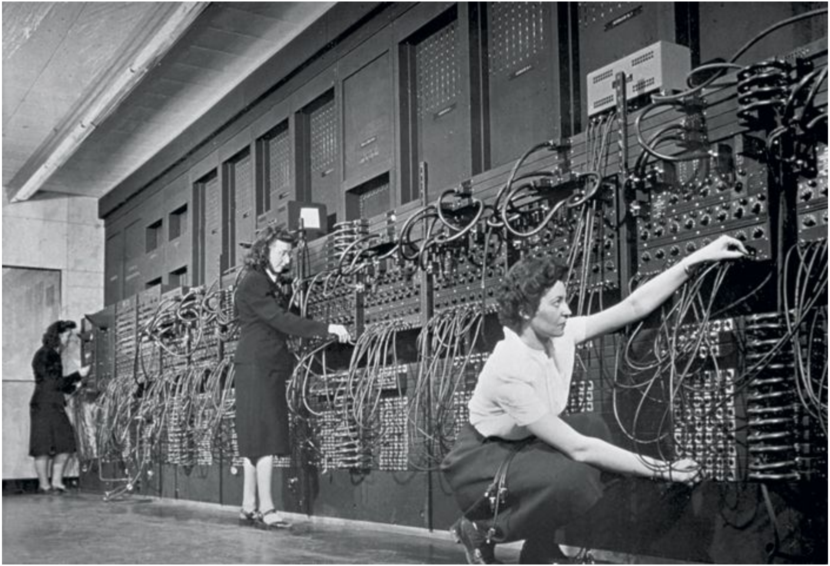


Bitonic sort — depth 4



Manual vs automatic placement

ENIAC, 1946:



Manual vs automatic placement

- Programmers used to explicitly place computations in space.
- Mainstream programming *still* manually places in time.
- Sequential composition: crude placement tool.
- Threads: notationally clumsy & hard to manage correctly.
- If we relinquish control, automation can do better.