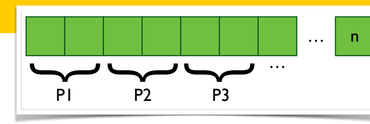
Utrecht University

B3CC: Concurrency

14: Data Parallelism (3)

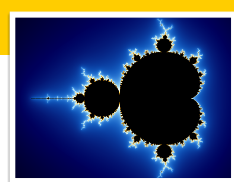
Ivo Gabe de Wolff

Recap



- Data parallelism: well understood approach to massive parallelism
  - Distributes the *data* over the different processing nodes
  - Executes the *same* computation on each of the nodes (threads)
  - Scales to very large numbers of processors
  - Conceptually simple: single thread of control

Recap



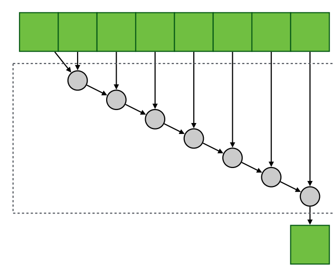
- So far our parallel patterns are *embarrassingly parallel*
  - Each operation is completely independent* from the computation in other threads
- But some collective operations deal with the data as a whole
  - The computation of each output element may depend on the results at other outputs (computed by other threads)
  - More difficult to parallelise!

```
__global__ void kernel( float* xs, float* ys, int n, … )
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if ( idx < n ) {
        // do something & communicate with others
    }
}
```


Fold

- Combine a collection of elements into a single value
  - A function combines elements pair-wise
  - Example: sum, minimum, maximum

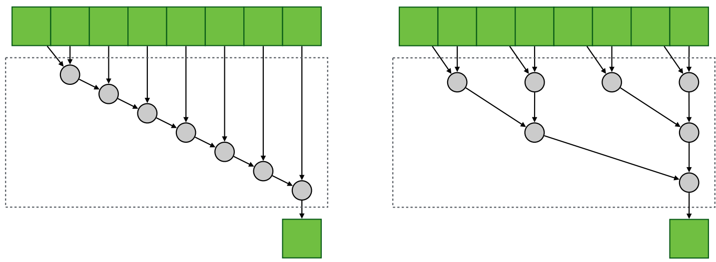
```
// fold1 (n > 0)
r = x[0];
for (i = 1; i < n; ++i)
  r = combine(r, x[i]);

// fold (n ≥ 0)
r = initial_value;
for (i = 0; i < n; ++i)
  r = combine(r, x[i]);
```



https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html#g:32

Fold

- Parallel reduction changes the order of operations
  - Number of operations remains the same, using $\lceil \log_2 N \rceil$ steps



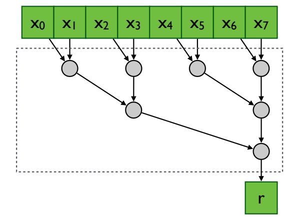
Sequential   Parallel

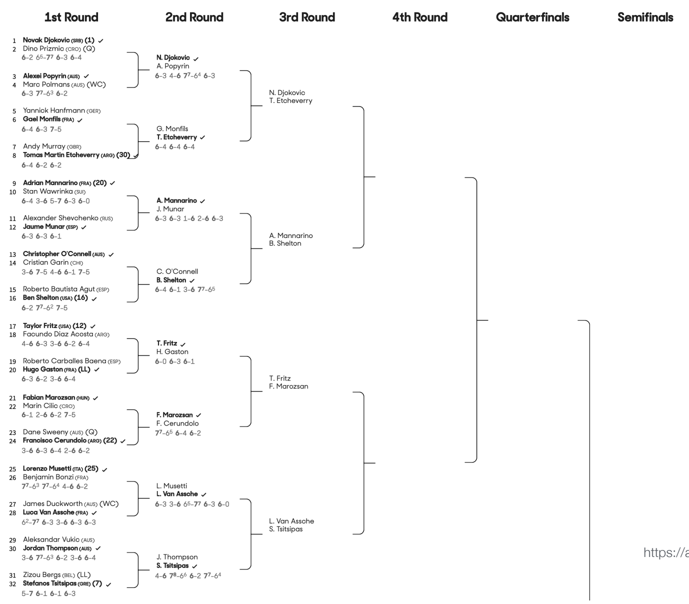
Fold

- Parallel reduction changes the order of operations
  - In order to do this, the combination function must be associative

$$
\begin{aligned}
r &= x_0 \otimes x_1 \otimes x_2 \otimes x_3 \otimes x_4 \otimes x_5 \otimes x_6 \otimes x_7 \\
&= ((((((x_0 \otimes x_1) \otimes x_2) \otimes x_3) \otimes x_4) \otimes x_5) \otimes x_6) \otimes x_7 \\
&= ((x_0 \otimes x_1) \otimes (x_2 \otimes x_3)) \otimes ((x_4 \otimes x_5) \otimes (x_6 \otimes x_7))
\end{aligned}
$$

  - Other optimisations are possible if the function is commutative, or the initial value is an identity element
  - In general difficult to automatically prove these properties for user defined functions





https://ausopen.com/draws#!mens-singles

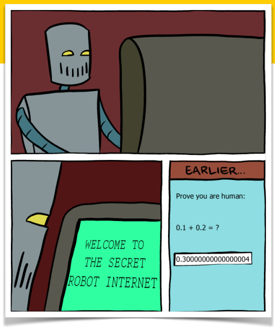
Fold in tournaments

- Australian Open has 128 participants
- Fold "computes" the best or maximum player
- Sequentially would take 127 days
  - Player 1 vs player 2, its winner vs player 3, that winner vs player 4, …
  - Assuming a person can only play one match per day
- With enough courts, this takes $\log_2(128) = 7$ days
- In reality, takes 15 days as the first rounds take multiple days

Associativity

- Sum works in parallel because addition is associative
  - Sequential: $(((x + y) + z) + w)$
  - Recursive: $((x + y) + (z + w))$
- Associative: change the position of the parentheses: $((x + y) + z) \equiv (x + (y + z))$
- Commutative: change the position of the variables: $x + y \equiv y + x$
  - Example:
    - Function composition is associative: $(f \cdot g) \cdot h \equiv f \cdot (g \cdot h)$
    - But not commutative: $(f \cdot g) \not\equiv (g \cdot f)$

Associativity



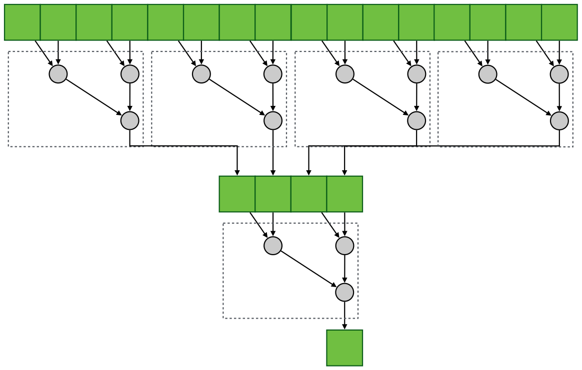
- "Best" in sports is probably not associative (nor deterministic)
- Strictly speaking, computer arithmetic is not associative
  - Integer arithmetic can over/underflow
  - Floating-point values have limited precision
  - Example: 7-*digit* mantissa

```
1234.567
  45.67844
   0.000400

1234.567 + 45.67844 = 1280.24544
         + 0.000400 = 1280.2454
                    = 1280.245

45.67844 + 0.000400 = 45.67884
         + 1234.567 = 1280.24584
                    = 1280.256
```

http://www.smbc-comics.com/comic/2013-06-05
https://en.wikipedia.org/wiki/Kahan_summation_algorithm

Fold

- In practice, the input is split into multiple tiles (chunks)
- The tiles are distributed over the available cores (for CPUs) or streaming multiprocessors (GPUs)
- The results per tile are then reduced
  - With a sequential fold, or recursively with a parallel fold



https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Fold

- Reduction happens on multiple levels in the hardware
- For a GPU:
  - Each thread handles multiple elements, with a sequential loop
  - Each warp reduces the values of its threads
  - Each thread block reduces the values of its warps and writes the results to global memory
  - In a separate kernel, we reduce the results of all thread blocks
- For a CPU:
  - Each SIMD lane …
  - Each thread …
  - Afterwards, reduce the results of all threads

Example: dot product

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n-1} a_i b_i$$

- The vector dot-product operation pair-wise multiplies the elements of two vectors, and then sums the result
  - A combination of `zipWith` followed by a `fold`
  - These operations can be *fused* to avoid storing the intermediate result
  - Array fusion is an important optimisation for collection-based programming models (c.f. loop fusion)

Scan

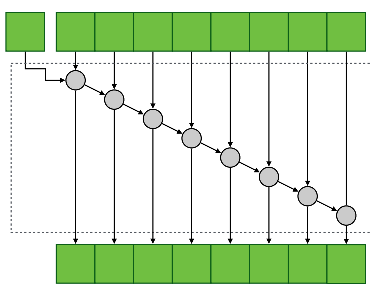
- Similar to reduce, but produces all partial reductions of the input
  - An important building-block in many parallel algorithms
    - Sorting algorithms, lexical comparison of strings, lexical analysis (parsing), evaluating polynomials, adding multi-precision numbers…
  - Trickier to parallelise than reduce
  - Two (main) variants: inclusive and exclusive
- Scan is an important building block in many parallel algorithms

https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html#g:35

Scan

- Two variants: inclusive and exclusive
  - Inclusive scan includes the current element in the partial reduction
  - Exclusive scan includes all prior elements

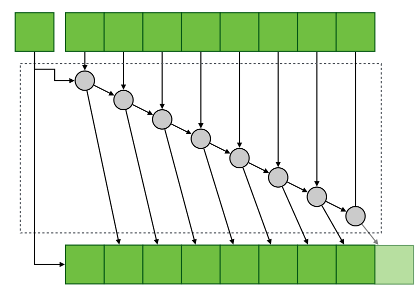
```
// inclusive: scanl1
r = initial_value;
for (i = 0; i < n; ++i) {
  r    = combine(r, x[i]);
  y[i] = r;
}
```



Scan

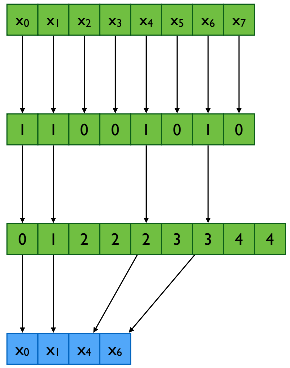
- Two variants: inclusive and exclusive
  - Inclusive scan includes the current element in the partial reduction
  - Exclusive scan includes all prior elements

```
// exclusive: scanl
r = initial_value;
for (i = 0; i < n; ++i) {
  y[i] = r;
  r    = combine(r, x[i]);
}
// optionally: y[i] = r;
```



Example: filter (compact)

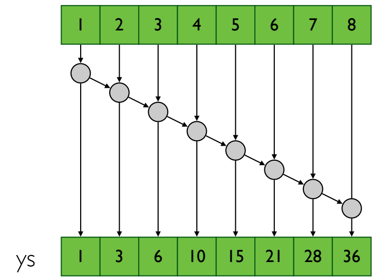
- Return only those elements of the array which pass a predicate
  1. *map* the predicate function over the values to determine which to keep
  2. exclusive *scan* the boolean flags to determine the output locations and number of elements to keep
  3. *permute* the values into the position given by (2) if (1) is true



https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html#g:31

Example: Integral Image

- Consider this inclusive prefix sum
  - We can use this result to calculate the sum of any interval of the input:
    `sum [3..6] = ys[5] - ys[1] = 21 - 3 = 18`



Example: Integral Image

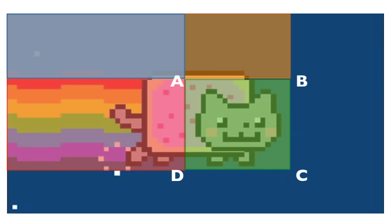
- This idea extends to two (or more) dimensions
  - Known as the integral image or summed area table

$$I(x,y) = \sum_{v=0}^{y} \sum_{u=0}^{x} i(u,v)$$

  - Suppose I want to find the sum of the green region:

$$I_{ABCD} = I_C - I_D - I_B + I_A$$

  - Can be used to implement a box filter in constant time
  - Key component of the Viola-Jones face recognition algorithm



https://youtu.be/uEJ71VlUmMQ

Scan

- In the prefix sum we produce all partial reductions of the input
  - That is, the reduction of every prefix

```
         input = [3,4, 4, 4, 4, 3, 5, 4, 5]
scanl1 (+) input = [3,7,11,15,19,22,27,31,36]
```

  - The prefix sum you might also think of as a cumulative sum
  - Variations for inclusive, exclusive, left, right, product, conjunction…
  - Sequential calculation is a single sweep of n-1 additions

```
for (i = 1, i < n; ++i)
    A[i] = A[i] + A[i-1]
```

https://en.wikipedia.org/wiki/Prefix_sum

Scan

- Example: how to parallelise prefix sum

```
   input: [3,4, 4, 4, 4, 3, 5, 4, 5]
expected: [3,7,11,15,19,22,27,31,36]
```

- Split the data over two processors and perform a prefix sum individually on each part:

```
             split: [3,4, 4, 4, 4] ¦ [3,5, 4, 5]
left/right result: [3,7,11,15,19] ¦ [3,8,12,17]
                         P1              P2
```

- The left part looks correct, but every element in the right part needs to be incremented by 19
- Luckily, this is the final result of the left side, which we just computed!

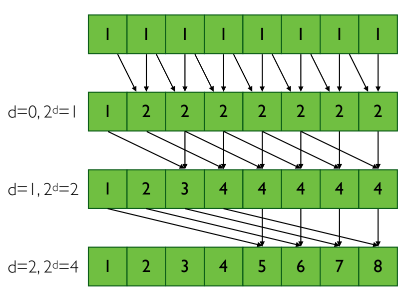
Scan

- Parallel scan split into tiles is classically done in three phases:
  1. Upsweep: Break the input into equally sized tiles, and reduce each tile
  2. Perform an exclusive scan of the reduction values
  3. Downsweep: Perform a scan of each tile, using the per-tile carry-in values computed in step 2 as the initial value

Scan

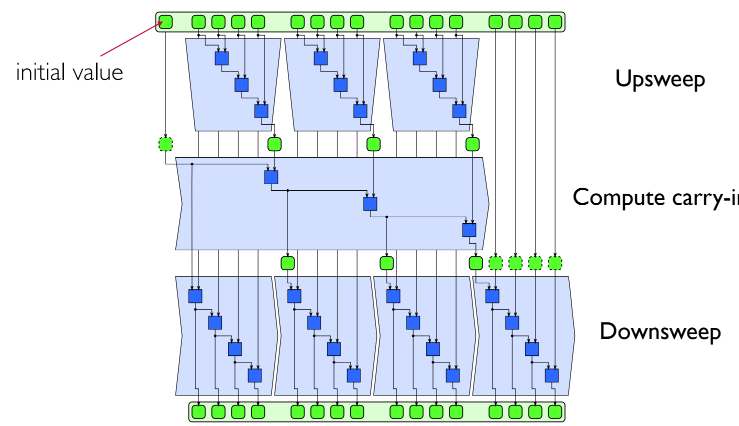
- Example: how to parallelise prefix sum (per-tile)
  - Here computed in SIMD (e.g. in a warp on the GPU)
  - Parallel scan [again] changes the order of operations

```
for ( d = 0
    ; d < log₂ N;
    ; d ++ )
{
  int offset = 2ᵈ;
  if (i ⩾ offset) // parallel
    x[i] = x[i-offset] + x[i];
}
```

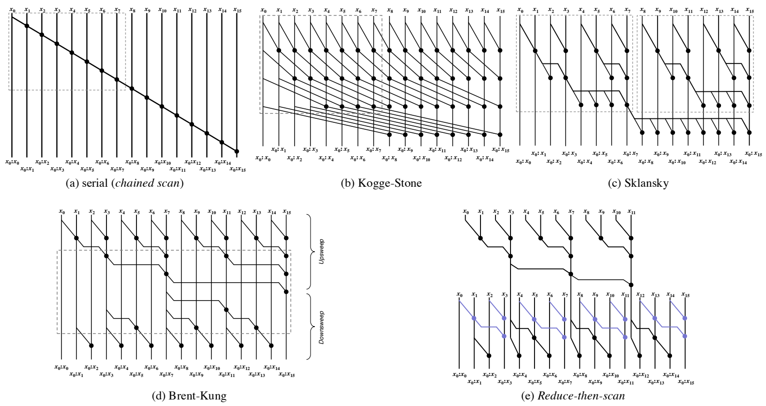


Scan

- Three-phase tiled implementation of inclusive scan:



Scan



(a) serial (*chained scan*) (b) Kogge-Stone (c) Sklansky (d) Brent-Kung (e) *Reduce-then-scan*

Three-phase scans on GPUs

- Scans are (or used to be) implemented via three phases on GPUs
  - Kernel 1 performs a fold per block
  - Kernel 2 scans over the results per block (using a single thread block)
  - Kernel 3 performs a scan per block, using the prefix of that block computed in kernel 2
- Synchronization between blocks happens by splitting the program in multiple kernels
  - Kernel 2 only starts when all thread blocks of kernel 1 have finished
- It is advised to not perform synchronization between thread blocks within the same kernel
  - But…

Chained scans on GPUs

- Chained scans use only one kernel, and do synchronize within the kernel
  - Each thread block does the following:
    - Read a tile of the array
    - Fold
    - Wait on prefix of previous tile
    - Share own prefix
    - Scan
  - Three-phase scans typically split the input in a fixed number of blocks,
    chained scans use fixed-size blocks as the data should fit in the registers of the threads of a thread block.

Chained scans on GPUs

- Chained scans go against the advice of independent thread blocks
- You have to be careful:
  - Don't use the hardware scheduler - implement your own scheduling of thread blocks
  - Prevent memory reordering
  - Waiting on the prefix of the previous block can be a significant bottleneck
    - The *Single-pass Parallel Prefix Scan with Decoupled Look-back* optimizes this
- Chained may be faster than three-phase scans
  - as they read the input once instead of twice

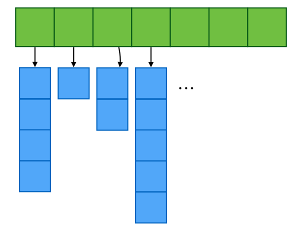
Flat data parallelism

- Widely used, well understood & supported approach to massive parallelism
  - Single point of concurrency
  - Easy to implement
  - Good cost model (work & span)
  - BUT! the "something" has to be sequential

```
__global__ void kernel( float* xs, float* ys, int n, … )
{
   int idx = blockDim.x * blockIdx.x + threadIdx.x;
   if ( idx < n ) {
      // do something sequentially
      // but can not launch further parallel work!
   }
}
```

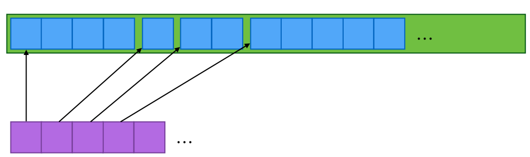

Nested data parallelism

- Main idea: allow the "something" to also be parallel
  - Now the parallelism structure is recursive and unbalanced
  - Still a good cost model
  - Wider range of applications: sparse arrays, adaptive methods (Barnes-Hut), divide and conquer (quicksort, quickhull), graph algorithms (shortest path, spanning tree)



Nested data parallelism

- The flattening transformation
  - Concatenate the subarrays into one big flat array
  - Operate in parallel on the big array
  - A *segment descriptor* keeps track of where the sub-arrays begin
- Example: given an array of nodes in a graph, compute an array of their neighbors
  - For instance in findRequests for Delta-stepping
- The `scan` operation gives us a way to do this



Segmented scan

- We can also create *segmented* versions of collective operations like scan
  - Generalises scan to perform separate parallel scans on arbitrary contiguous partitions (segments) of the input vector
  - In particular useful for sparse and irregular computations

values	3	1	7	0	4	1	6	3
segment descriptor	1	0	1	0	0	1	0	1
scan_seg	3	4	7	7	11	1	7	3

  - Can be implemented via operator transform:

$$(f_x, x) \oplus^s (f_y, y) = (f_x | f_y,\ \text{if } f_y \text{ then } y \text{ else } x \oplus y)$$

Segmented scan

- Lift a binary operator to a segmented version:
  - Can be implemented via operator transform
  - The lifted operator should be associative!
    - Concretely, if ⊕ is associative, then $\oplus^s$ should also be associative

$$(f_x, x) \oplus^s (f_y, y) = (f_x | f_y, \text{ if } f_y \text{ then } y \text{ else } x \oplus y)$$

```
segmented
    :: Elt a
    => (Exp a -> Exp a -> Exp a)
    -> (Exp (Bool, a) -> Exp (Bool, a) -> Exp (Bool, a))
segmented op (T2 fx x) (T2 fy y)
  = T2 ( fx || fy )
       ( fy ? (y, op x y) )
```


Segment descriptors

- Segment descriptors *describe* where *segments* start, via
  - Segment lengths, or
  - Head flags
- Create the *head flags* array from segment lengths
  - The segment descriptor tells us the length of each segment
  - To use the operator from the previous slide, we need to convert this into a representation the same size as the input, with a `True` value at the start of each segment and `False` otherwise

```
mkHeadFlags :: Acc (Vector Int) -> Acc (Vector Bool)
mkHeadFlags seg =
  let
      T2 offset len = scanl' (+) 0 seg
      falses        = fill (I1 (the len)) False_
      trues         = fill (shape seg)    True_
  in permute const falses
       (\ix -> Just_ (I1 (offset!ix))) trues
```


Segmented scan

- What about other flavours of scan?
  - This approach works directly for *inclusive* segmented scan
  - The exclusive version is similar, but needs to fill in the initial element and take care of (multiple consecutive) empty segments

Conclusion

- Fold (reduction) and scan (prefix sum) can be executed in parallel
  - if the operator is associative: $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
- Prefix sum is a useful application in many (parallel) programming problems
  - Use to compute the book-keeping information required to execute nested data-parallel algorithms on flat data-parallel hardware (e.g. GPUs)


tot ziens
Photo by Anusha Barwa