OpenCL Day 3: Basic Block of Data-Parallel Programming and Applications

Cosmin Oancea and Troels Henriksen

February 4, 2019

Course Contents

Basic Blocks of Parallel Programming: Map

map : $(\alpha \to \beta, [\alpha]) \quad \to \quad [\beta]$ has *inherently parallel semantics*.

Applies a function to every element of the input array producing an array of equal length.

$$
\begin{array}{lllllll}
X = & \text{map}(\, f, & [& a_1, & a_2, & .., & a_n &]) \\
 & & & \downarrow & \downarrow & & \downarrow & \\
X \equiv & & [& f(a_1), & f(a_2), & .., & f(a_n) &]
\end{array}
$$

Similar: `map2(f,[`a_1`,...,`a_n`],[`b_1`,...,`b_n`])` $\equiv$ `[f(`a_1`,`b_1`),...,f(`a_n`,`b_n`)]`

Basic Blocks of Parallel Programming: Map

map : $(\alpha \to \beta, [\alpha]) \quad \to \quad [\beta]$ has *inherently parallel semantics*.

Applies a function to every element of the input array producing an array of equal length.

$$
\begin{array}{llcccccl}
X = & \text{map}(f, & [& a_1, & a_2, & .., & a_n &]) \\
 & & & \downarrow & \downarrow & & \downarrow & \\
X \equiv & & [& f(a_1), & f(a_2), & .., & f(a_n) &]
\end{array}
$$

Similar: `map2(f,[`a_1`,...,`a_n`],[`b_1`,...,`b_n`])` $\equiv$ `[f(`a_1`,`b_1`),...,f(`a_n`,`b_n`)]`

Map Fusion: `map(f, map(g, A))` $\equiv$ `map(f o g, A)`

$$
\begin{array}{lcllllll}
A & \equiv & [& a_1, & a_2, & \ldots, & a_n &] \\
X = \text{map}(g, A) & \equiv & [& g(a_1), & g(a_2), & \ldots, & g(a_n) &] \\
Y = \text{map}(f, X) & \equiv & [& f(g(a_1)), & f(g(a_2)), & \ldots, & f(g(a_n)) &] \\
\text{map}(f \circ g, A) & \equiv & [& f(g(a_1)), & f(g(a_2)), & \ldots, & f(g(a_n)) &]
\end{array}
$$

Fusion is a very important optimizations; saves bandwidth!

Basic Blocks of Parallel Programming: Reduce

reduce : $((\alpha, \alpha) \rightarrow \alpha),\ \alpha, [\alpha]) \quad \rightarrow \quad \alpha$

reduce$(\ \odot,\ 0_{\odot},\ [a_1, a_2, \ldots, a_n]) \equiv 0_{\odot}\ \odot\ a_1\ \odot a_2\ \odot \ldots \odot\ a_n$

where $\odot$ is an associative binary operator (otherwise bug!)

$0_{\odot}$ is the neutral element of the monoid induced by $\odot$

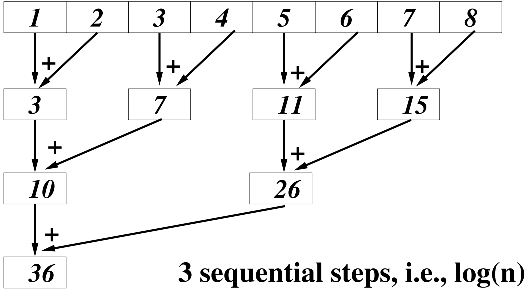
Basic Blocks of Parallel Programming: Reduce

$\text{reduce} : ((\alpha, \alpha) \to \alpha),\ \alpha, [\alpha]) \ \to \ \alpha$

$\text{reduce}(\ \odot,\ 0_{\odot},\ [a_1, a_2, \ldots, a_n]) \equiv 0_{\odot}\ \odot\ a_1\ \odot a_2\ \odot \ldots \odot\ a_n$

where $\odot$ is an associative binary operator (otherwise bug!)

$0_{\odot}$ is the neutral element of the monoid induced by $\odot$



Build programs by combining map, reduce and other such operators.

Trivial Examples of Map-Reduce Programming

Small Exercise: write a function that receive as parameters a predicate `p` $: \alpha \to$ `bool` and an array A, and that results in `true` if all elements satisfy `p` and in `false` otherwise.

Try to write (i) a divide-and-conquer and (ii) a map-reduce implementation.

Trivial Examples of Map-Reduce Programming

Small Exercise: write a function that receive as parameters a predicate `p` $: \alpha \to$ `bool` and an array A, and that results in `true` if all elements satisfy `p` and in `false` otherwise.

Try to write (i) a divide-and-conquer and (ii) a map-reduce implementation.

```
all(p, [ ])  = True
all(p, [x])  = p(x)
all(p, x++y) = all(p, x) &&
               all(p, y)
```

```
all(p, x) =
    y = map(p, x)
    z = reduce(&&, true, y)
    return z
```

Trivial Examples of Map-Reduce Programming

Small Exercise: write a function that receive as parameters a predicate `p` $: \alpha \rightarrow$ `bool` and an array A, and that results in `true` if all elements satisfy `p` and in `false` otherwise.

Try to write (i) a divide-and-conquer and (ii) a map-reduce implementation.

```
all(p, [ ])  = True
all(p, [x])  = p(x)
all(p, x++y) = all(p, x) &&
               all(p, y)
```

```
all(p, x) =
    y = map(p, x)
    z = reduce(&&, true, y)
    return z
```

++ denotes array concatenation.

A well define divide and conquer implementation requires that any split of the input array into `x++y` gives the same result.

(This is equivalent to the requirement that the binary operator of reduction is associative.)

Under this conditions, the two are equivalent: if you can write one, then you can derive the other (list homomorphic $\equiv$ map-reduce).

Asymptotic Work and Depth; Brent Lemma

Assuming an infinity of processors:

- Work Complexity $W(n)$: is the total # of ops performed,
- Depth/Step Complexity $D(n)$: is the # of sequential steps.

- A parallel implem is **work efficient** *iff* its work complexity is equal to the one of the golden sequential implem.

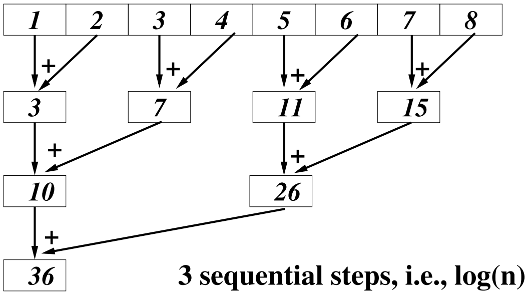
- Work and Depth are good high-level approximations;
- If we know the work and depth asymptotic for a program, Brent Theorem offers good complexity bounds for a PRAM.

Theorem (Brent Theorem)

An algorithm of depth $D(n)$ and work $W(n)$ can be simulated on a P-processor PRAM in time complexity T such that:

$$\frac{W(n)}{P} \leq T \leq \frac{W(n)}{P} + D(n)$$

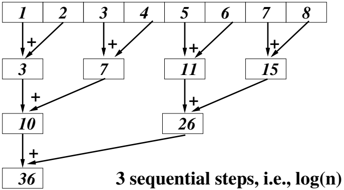
Applying Brent Lemma to Map-Reduce Computation



Reducing an array of length n with n/2 processors requires:

- work $W(n) = n$ and
- depth $D(n) = lg\ n$, i.e., number of sequential steps.
- Brent Theorem states the bounds for optimal runtime T^{opt}:
  $\frac{W(n)}{P} \leq T^{opt} \leq \frac{W(n)}{P} + D(n)$

Applying Brent Lemma to Map-Reduce Computation



Reducing an array of length n with n/2 processors requires:

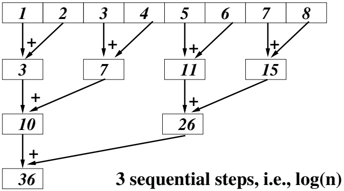
- $W(n) = n, D(n) = lg\ n$.
- Optimal runtime T^{opt} bounds: $\frac{W(n)}{P} \leq T^{opt} \leq \frac{W(n)}{P} + D(n)$

An optimized map-reduce computation can be implemented as:

- splits the input array into P subarrays, each containing about the same number of elements;
- perform the computation sequentially on each subarray, but in parallel across subarrays;
- reduce the P per-processor results.

This leads to optimal runtime on P processors:

Applying Brent Lemma to Map-Reduce Computation



Reducing an array of length n with n/2 processors requires:

- $W(n) = n, D(n) = lg\ n$.
- Optimal runtime T^{opt} bounds: $\frac{W(n)}{P} \leq T^{opt} \leq \frac{W(n)}{P} + D(n)$

An optimized map-reduce computation can be implemented as:

- splits the input array into P subarrays, each containing about the same number of elements;
- perform the computation sequentially on each subarray, but in parallel across subarrays;
- reduce the P per-processor results.

This leads to optimal runtime on P processors: $O(\frac{n}{P} + lg\ P)$

This kind of chunking is often referred to as:
“efficient sequentialization of excess parallelism”!

Almost Homomorphisms (Gorlatch)

"Systematic Extraction and Implementation of Divide-and-Conquer Parallelism", Sergei Gorlatch, 1996.

Intuition: a non-homomorphic function g can be sometimes lifted to a homomorphic one f, by computing a baggage of *extra info*.

The initial fun obtained by projecting the homomorphic result:
$g = \pi \circ f$

Maximum-Segment Sum Problem (mssp**)**:
Given a list of integers, find the contiguous segment of the list whose members have the largest sum among all such segments. The result is only the maximal sum (not the segment's members). For simplicity lets assume we are interested only in **positive sums**.

E.g., `mss` [1, -2, 3, 4, -1, 5, -6, 1] = 11
(the corresponding segment is [3, 4, -1, 5]).

Maximum Segment Sum (MSSP): Preliminary Reasoning

Maximum-Segment Sum Problem (mss**)**:
Given a list of integers, find the contiguous segment of the list whose members have the largest sum among all such segments. The result is only the maximal sum (not the segment's members). For simplicity lets assume we are interested only in **positive sums**.

A first straightforward/naive attempt:

```
mss [ ]      = 0
mss [a]      = a ↑ 0  //↑ denotes Max
mss (x ++ y) = mss(x) ??? mss(y)
```

Maximum Segment Sum (MSSP): Preliminary Reasoning

Maximum-Segment Sum Problem (mss):
Given a list of integers, find the contiguous segment of the list whose members have the largest sum among all such segments. The result is only the maximal sum (not the segment's members). For simplicity lets assume we are interested only in **positive sums**.

A first straightforward/naive attempt:

```
mss [ ]        = 0
mss [a]        = a ↑ 0   //↑ denotes Max
mss (x ++ y) = mss(x) ??? mss(y)
```

x | **y**

1	−2	3	4	−1	5	−6	1

mss1 = 7 | **mss2 = 5**

Which case is problematic?

How to combine mss1 and mss2?

mss1 + mss2 = 12 Incorrect!

max(mss1,mss2) = 7 Incorrect!

Maximum Segment Sum (MSSP): Preliminary Reasoning

Maximum-Segment Sum Problem (mss**)**:
Given a list of integers, find the contiguous segment of the list whose members have the largest sum among all such segments. The result is only the maximal sum (not the segment's members). For simplicity lets assume we are interested only in **positive sums**.

A first straightforward/naive attempt:

```
mss [ ]       = 0
mss [a]       = a ↑ 0  //↑ denotes Max
mss (x ++ y) = mss(x) ??? mss(y)
```

x				y			
1	−2	3	4	−1	5	−6	1

mss1 = 7　　**mss2 = 5**

Which case is problematic?

Answer: when the segment of interest lies partly in x and partly in y!

How to combine mss1 and mss2?

mss1 + mss2 = 12 **Incorrect!**

max(mss1,mss2) = 7 **Incorrect!**

Maximum Segment Sum (MSSP): A Better Reasoning

The problematic case is when the segment of interest lies partly in x and partly in y!

We need to compute extra information:

Maximum Segment Sum (MSSP): A Better Reasoning

The problematic case is when the segment of interest lies partly in x and partly in y!

We need to compute extra information:

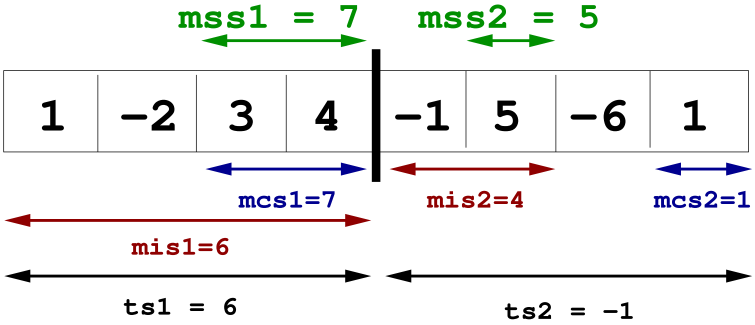
- maximum concluding segment
- maximum initial segment
- total segment sum

Maximum Segment Sum (MSSP): A Better Reasoning

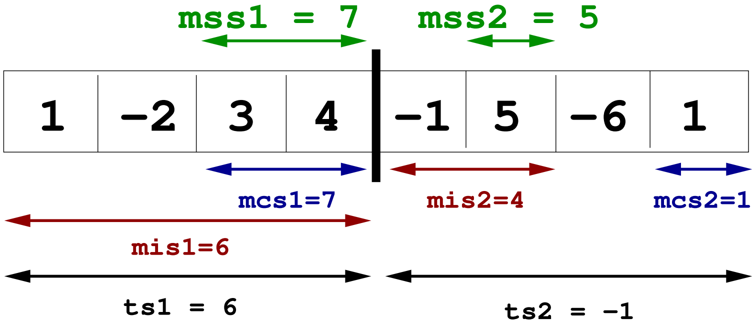
The problematic case is when the segment of interest lies partly in x and partly in y!

We need to compute extra information:

- maximum concluding segment
- maximum initial segment
- total segment sum



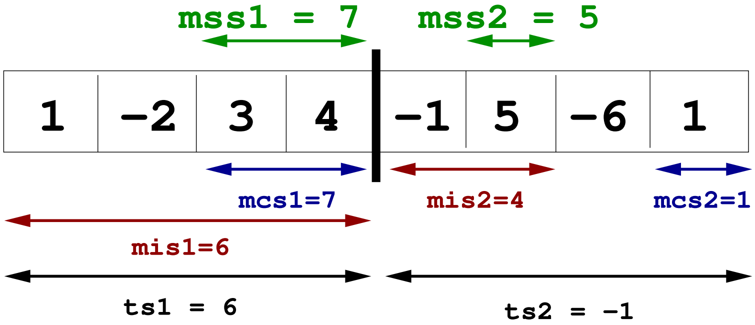
MSSP: Deriving the Implementation



Lets compute the `mis`, `mcs`, `mss`, and `ts` for the result of the two concatenating segments. ↑ denotes `max`.

```
mis =
```

MSSP: Deriving the Implementation

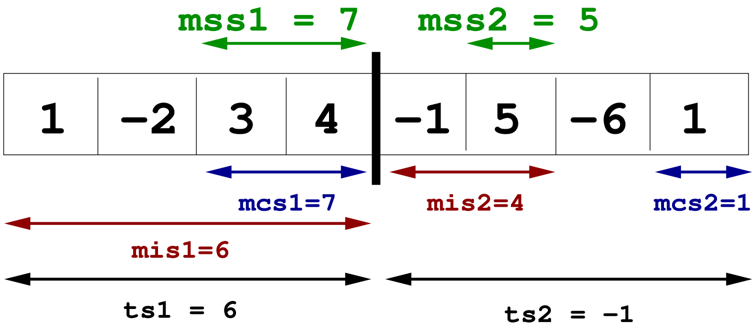


Lets compute the `mis`, `mcs`, `mss`, and `ts` for the result of the two concatenating segments. ↑ denotes `max`.

```
mis = mis1 ↑ (ts1 + mis2)

mcs =
```

MSSP: Deriving the Implementation



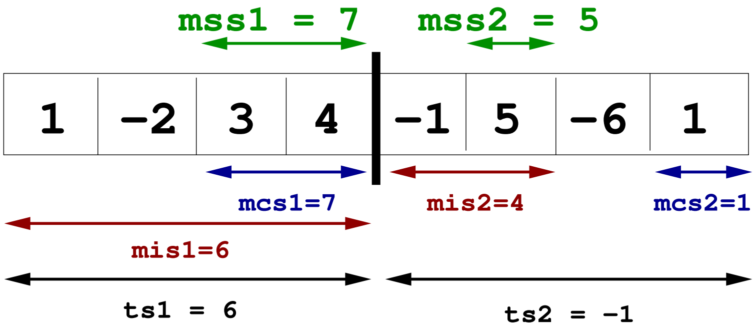
Lets compute the `mis`, `mcs`, `mss`, and `ts` for the result of the two concatenating segments. ↑ denotes `max`.

```
mis = mis1 ↑ (ts1 + mis2)

mcs = mcs2 ↑ (mcs1 + ts2)

mss =
```

MSSP: Deriving the Implementation



Lets compute the `mis`, `mcs`, `mss`, and `ts` for the result of the two concatenating segments. ↑ denotes `max`.

```
mis = mis1 ↑ (ts1 + mis2)
mcs = mcs2 ↑ (mcs1 + ts2)
mss = mss1 ↑ mss2 ↑ (mcs1 + mis2)
ts  = ts1 + ts2
```

MSSP: Map-Reduce C-ish Pseudocode

```
// x ↑ y ≡ (x >= y) ? x : y
Qtup ⊙ (Qtup x, Qtup y) {
    Qtup r;
    r.mss = x.mss ↑ y.mss ↑
            (x.mcs + y.mis);
    r.mis = x.mis ↑ (x.ts + y.mis);
    r.mcs = (x.mcs + y.ts) ↑ y.mcs;
    r.ts  = x.ts + y.ts;
    return r;
}

Qtup f(int x) {
    Qtuple r;
    int x0 = x ↑ 0;
    r.mss = x0; r.mis = x0;
    r.mcs = x0; r.ts = x;
    return r;
}
```

```
typedef struct msict {
  int mss;
  int mis;
  int mcs;
  int ts;
} Qtup;

int mss(int* xs) {
    Qtup ne, res;
    ne.mss = 0; ne.mis = 0;
    ne.mcs = 0; ne.ts  = 0;

    Qtup* ys = map(f, xs);
    res = reduce(⊙, ne, ys);
    return res.mss;
}
```

The baggage: 3 extra integers (`misx`, `mcsx`, `tsx`) and a constant number of integer operations per communication stage.

For performance: array `ys` should not be manifested in memory; fuse the map with the `reduce` operations.

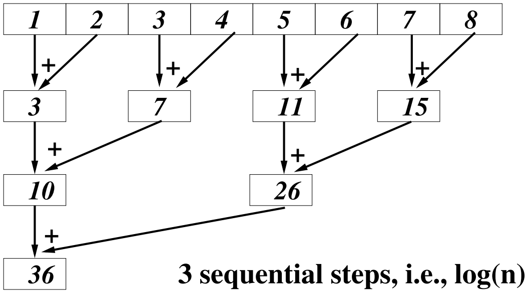
Exercise 1: OpenCL Implementation of MSSP

- For this exercise, you will implement MSSP as an OpenCL program.
- As inspiration, we will first study four OpenCL implementations of *summing an integer array*.

$$\sum_{i<n} x[i]$$

Binary Tree Reduction

The idea: each thread reads two neighbouring elements, adds them together, and writes one element. This halves the array in size. Continue until only a single element is left.



- Each level becomes a kernel invocation, with number of threads equal to half the number of array elements.
- $O(n)$ work and $O(\log(n))$ span (optimal).
- **Why is this not efficient?**

Improving the Tree Reduction

The idea: instead of shrinking the array by a factor of two for each level, shrink it by the workgroup size.

- Same asymptotic performance.
- Avoids kernels with very few threads. E.g with workgroup size 256: $10000000 \rightarrow 39063 \rightarrow 153 \rightarrow 1$.

Improving the Tree Reduction

The idea: instead of shrinking the array by a factor of two for each level, shrink it by the workgroup size.

- Same asymptotic performance.
- Avoids kernels with very few threads. E.g with workgroup size 256: $10000000 \rightarrow 39063 \rightarrow 153 \rightarrow 1$.

Implementation	$n = 1000$	$n = 1000000$
Tree reduction	$77\mu s$	$363\mu s$
Group reduction	$17\mu s$	$179\mu s$

Applying Brent's Lemma

The idea: instead of letting the thread count depend on the input size, always launch the same number of threads, and have each thread perform an efficient sequential summation of a *chunk* of the input.

- GPUs have a maximum (hardware/problem-dependent) capacity for exploiting parallelism. Beyond that limit, parallelism is at best worthless, and usually comes with overhead (e.g. excessive synchronisation).
- *A straightforward implementation of this idea only works if the operator is commutative!*

Applying Brent's Lemma

The idea: instead of letting the thread count depend on the input size, always launch the same number of threads, and have each thread perform an efficient sequential summation of a *chunk* of the input.

- GPUs have a maximum (hardware/problem-dependent) capacity for exploiting parallelism. Beyond that limit, parallelism is at best worthless, and usually comes with overhead (e.g. excessive synchronisation).
- *A straightforward implementation of this idea only works if the operator is commutative!*

Implementation	$n = 1000$	$n = 1000000$
Tree reduction	$77\mu s$	$363\mu s$
Group reduction	$17\mu s$	$179\mu s$
Chunked reduction	$70\mu s$	$103\mu s$

Using Atomics

The idea: GPUs have special hardware support for performing certain memory updates atomically. In OpenCL, this is exposed through *atomic operations*.

```
int atomic_add(volatile __global int *p,
               int val)
```

- Concise parallel reduction: each thread reads an element and uses `atomic_add()` to update the same location in memory.
- **Why is this slow for large inputs?**

Using Atomics

The idea: GPUs have special hardware support for performing certain memory updates atomically. In OpenCL, this is exposed through *atomic operations*.

```
int atomic_add(volatile __global int *p,
               int val)
```

- Concise parallel reduction: each thread reads an element and uses `atomic_add()` to update the same location in memory.
- **Why is this slow for large inputs?**

Implementation	$n = 1000$	$n = 1000000$
Tree reduction	$77\mu s$	$363\mu s$
Group reduction	$17\mu s$	$179\mu s$
Chunked reduction	$70\mu s$	$103\mu s$
Atomics	$8\mu s$	$1278\mu s$

For MSSP

Implement three versions:

1. Tree reduction
2. Group reduction
3. Chunked reduction

Hints:

- Instead of a `struct Qtup`, you may want to use `int4` (inside the kernels) and `cl_int4` (in host code).
- The MSSP operator is *not* commutative, so you will need to use a sliding window approach for the chunked reduction.
- For the chunked reduction, you can fuse the `map` function into the reduction itself. For the others, a separate pass is done.

Course Contents

Scan: A Basic Block of Data-Parallel Programming

Scan is also known as parallel prefix sum:

- computes all partial prefixes of an array;
- similar type with reduce, except that it returns an array;

- exclusive scan: result array starts with the neutral element;
- inclusive scan: starts with the first element of the input array;

- Inclusive scan is slightly more useful than exclusive scan.

Scan: A Basic Block of Data-Parallel Programming

Scan is also known as parallel prefix sum:

- computes all partial prefixes of an array;
- similar type with reduce, except that it returns an array;

- exclusive scan: result array starts with the neutral element;
- inclusive scan: starts with the first element of the input array;

- Inclusive scan is slightly more useful than exclusive scan.

$\mathtt{scan}^{exc}: \; ((\alpha,\alpha) \to \alpha),\; \alpha,\; [\mathtt{n}]\alpha \;) \to [\mathtt{n}]\alpha$

$\mathtt{scan}^{exc}(\odot, \mathtt{e}, [\mathtt{x}_1, \ldots, \mathtt{x}_n]) = [\mathtt{e}, \mathtt{e}\odot\mathtt{x}_1, \ldots, \mathtt{e}\odot\mathtt{x}_1\odot\ldots\mathtt{x}_{n-1}]$

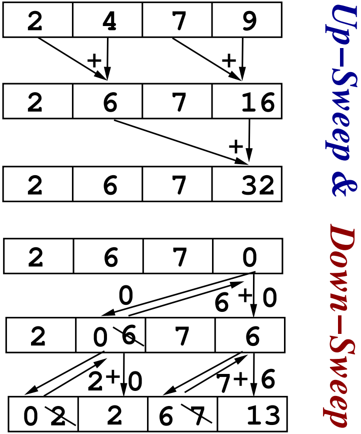
i.e., $\mathtt{e}:\alpha,\; \mathtt{x}_i:\alpha,\; \forall i$, and $\odot:(\alpha,\alpha) \to \alpha$.

$\mathtt{scan}^{inc}: \; ((\alpha,\alpha) \to \alpha),\; \alpha,\; [\mathtt{n}]\alpha \;) \to [\mathtt{n}]\alpha$

$\mathtt{scan}^{inc}(\odot, \mathtt{e}, [\mathtt{x}_1, \ldots, \mathtt{x}_n]) = [\mathtt{x}_1, \; \mathtt{x}_1\odot\mathtt{x}_2, \ldots, \mathtt{x}_1\odot\ldots\odot\mathtt{x}_n]$

i.e., $\mathtt{e}:\alpha,\; \mathtt{x}_i:\alpha,\; \forall i$, and $\odot:(\alpha,\alpha) \to \alpha$.

Parallel Exclusive Scan with Associative Operator $\oplus$

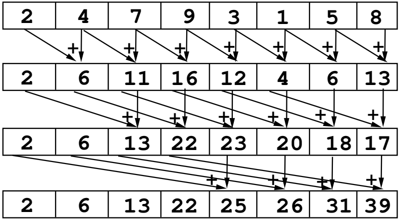


Two Steps:

- Up-Sweep: similar with reduction
- Root is replaced with neutral element.
- Down-Sweep:
  - the left child sends its value to parent and updates its value to that of parent.
  - the right-child value is given by $\oplus$ applied to the left-child value and the (old) value of parent.
  - note that the right child is in fact the parent, i.e., in-place algorithm.

Scan's Work and Depth: $D(n) = \Theta(lg\ n), W(n) = \Theta(n)$

Wavefront-Level Inclusive Scan for GPUs



Input: array A of $n=2^k$ elements of type α

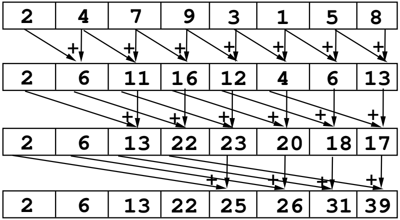
$\oplus : (\alpha, \alpha) \rightarrow \alpha$ associative

Output: B = $[a_1, a_1 \oplus a_2, \ldots, \oplus_{j=0}^{n-1} a_j]$

```
forall i = 0 : n-1 do
  B[i] ← A[i]
endfor
for d = 0 to k-1 do
  h = 2^d
  forall i = h to n-1 do
    B[i] ← B[i-h] ⊕ B[i]
  endfor
endfor
```

Offers better performance because it operates in one sweep rather than two!

Exercise 2: Intra-Wave Inclusive Scan Implementation



Input: array A of $n=2^k$ elements of type α

$\oplus : (\alpha, \alpha) \rightarrow \alpha$ associative

Output: B = $[a_1, a_1 \oplus a_2, \ldots, \oplus_{j=0}^{n-1} a_j]$

```
forall i = 0 : n-1 do
  B[i] ← A[i]
endfor
for d = 0 to k-1 do
  h = 2^d
  forall i = h to n-1 do
    B[i] ← B[i-h] ⊕ B[i]
  endfor
endfor
```

- Open file ”scanapps.cl”, and implement function named ”incScanWave” (follow the instructions)
- Your `n = WAVE` and `k = lgWAVE`; Ignore the init. loop;
- Unroll the `for d` loop (`#pragma unroll`);
- loop `forall i = h to n-1` is implicit,
- it should be replace by a condition `if (i>=h) { ... }`,
- except that `i` is not exactly the thread id.
- Remember, you want to `scan` each wave, independently!

OpenCL Scan Implementation

BLACKBOARD!

- CPU stub is in `Day3-Exercises/ScanApps/scan.h`
- Hierarchical design:

1. sequentially scan ELEMS_PER_THREAD by each thread,
2. publish in local memory
3. scan at wavefront level
4. "update" the scan at workgroup level
5. Gather the last elements in the scan groups into a separate array, and scan that!
5. Virtualize steps [1-4] so that the scan at step 5 fits into one workgroup;
6. add element `i-1` of the result of step 5 to each element of workgroup `i` resulted from step 4!

- This requires 2 reads and 2 writes from/to global memory. Can you do it better?

Scatter: Parallel Write Operator

Scatter updates in parallel and in place an input array with a set of values at specified indices:

scatter : $([m]\alpha, [n]\text{int}, [n]\alpha) \rightarrow [m]\alpha$

```
A (data vector)   =[b0, b1, b2, b3]
I (index vector)  =[2,  4,  1,  -1]
X (input array)   =[a0, a1, a2, a3, a4, a5]
scatter X I A     =[a0, b2, b0, a3, b1, a5]
```

Scatter: Parallel Write Operator

Scatter updates in parallel and in place an input array with a set of values at specified indices:

scatter : $([m]\alpha, [n]\text{int}, [n]\alpha) \rightarrow [m]\alpha$

```
A (data vector)   =[b0, b1, b2, b3]
I (index vector)  =[2,  4,  1,  -1]
X (input array)   =[a0, a1, a2, a3, a4, a5]
scatter X I A     =[a0, b2, b0, a3, b1, a5]
```

scatter has $D(n) = \Theta(1)$ and $W(n) = \Theta(n)$,
i.e., requires n update operations (n is the size of I or A, not of X!).

replicate(n, v) creates an array of length n filled with element v!

Exercise 3: Partition2 Operator

Type and Semantics of Partition2:

$\texttt{partition2 : } (\alpha \rightarrow \texttt{Bool}, \texttt{[n]}\alpha) \rightarrow (\texttt{int}, \texttt{[n]}\alpha)$

Partition2 receives as input a predicate and an array and results in:

- an integer denoting the number of elements that succeed under predicate, tupled with
- a new array, having the same elements as the input array, but reordered such as the elements that succeed under the predicate occur before the others.
- The partial order of the elements that succeed/fail should be respected.

```
bool even(int v) { return (bool)(1 -(v&1)); }
partition2(even, [5, 4, 2, 10, 3, 7, 8] ) should result in
             (4, [4, 2, 10, 8, 5, 3, 7])
```

Exercise 3, Step 1: Partition2 High-Level Implem

```
bool even(int v) { return (bool)(1 -(v&1)); }
partition2(even, [5, 4, 2, 10, 3, 7, 8] ) should result in
             (4, [4, 2, 10, 8, 5, 3, 7])
```

Step 1: implement partition2 based on map, scan and scatter (?)

```
partition2(X: [n]i32): (i32, [n]i32) =
 cs = map(even, X);
 tfs= map (λc->if c then 1 else 0
          ,  cs);
```

Exercise 3, Step 1: Partition2 High-Level Implem

```
bool even(int v) { return (bool)(1 -(v&1)); }
partition2(even, [5, 4, 2, 10, 3, 7, 8] ) should result in
             (4, [4, 2, 10, 8, 5, 3, 7])
```

Step 1: implement partition2 based on map, scan and scatter (?)

```
partition2(X: [n]i32): (i32, [n]i32) =
 cs = map(even, X);
 tfs= map (λc->if c then 1 else 0
          ,  cs);
 isT= scan( (+), 0, tfs );
 i  = isT[n-1];

 ffs= map (λc->if c then 0 else 1
          ,  cs );
 tmps = scan( (+), 0, ffs );
 isF= map (λt -> t+i, tmps);

 inds=map3(λ(c,iT,iF) ->
              if c then iT-1 else iF-1
          , cs, isT, isF);
 res = scatter(scratch n, inds, X);
 return (i, res);
```

```
X    = [5, 4, 2, 10,3, 7, 8]
n    = 7
cs   = [F, T, T, T, F, F, T]
tfs  = [0, 1, 1, 1, 0, 0, 1]

isT  = [0, 1, 2, 3, 3, 3, 4]
i    = 4

ffs  = [1, 0, 0, 0, 1, 1, 0]
tmps= [1, 1, 1, 1, 2, 3, 3]
isF  = [5, 5, 5, 5, 6, 7, 7]

inds= [4, 0, 1, 2, 5, 6, 3]
X    = [5, 4, 2, 10,3, 7, 8]
Res  = [_, _, _, _, _, _, _]
-----------------------------
Res  = [4, 2, 10, 8,5, 3, 7]
```

Exercise 3: Implement Partition2 in OpenCL

- CPU stub is `Day3-Exercises/ScanApps/partition2.h`, function `runPartition`, but nothing to do there!

(1) In `Day3-Exercises/ScanApps/scanapps.cl`, implement from scratch kernel `mapPredPartKer` which is supposed to compute `tfs` and `ffs`.

(2) the two scans are already implemented for you!

(3) Then implement kernel `scatterPartKer` which is supposed to compute the last part! Note that `isT` and `isF` are the results of `scan`, i.e., `isF` is actually `tmps` in slides! It is Ok to redundantly compute the condition again.

What is a Segmented Scan?

What is a Segmented Scan?

Assume an irregular matrix (two-dimensional array), i.e., rows have different number of elements.

Segmented Scan Intuition: the operation that scans each row of an irregular matrix and returns the result.

Flat Representation of a 2D irregular array: flag + value arrays:

What is a Segmented Scan?

Assume an irregular matrix (two-dimensional array), i.e., rows have different number of elements.

Segmented Scan Intuition: the operation that scans each row of an irregular matrix and returns the result.

Flat Representation of a 2D irregular array: flag + value arrays:

```
[ [1, 2, 3]
, [4, 5, 6, 7, 8]
, [9, 10]
]
```

```
Flag Array:
[1, 0, 0, 1, 0, 0, 0, 0, 1, 0 ]
Value Array:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

The flag array marks with one (or something different than zero) the start of a row/segment, the other elements are zero.

The value array: flat sequence of elements.

$\texttt{sgmScan}^{inc/exc}\texttt{:}\ \texttt{((}\alpha\texttt{,}\alpha\texttt{)}\rightarrow\alpha\texttt{),}\ \alpha\texttt{,}\ \texttt{[n]int,}\ \texttt{[n]}\alpha\texttt{)}\ \rightarrow\ \texttt{[n]}\alpha$

```
sgmScan^inc((+), 0, [1, 0, 0, 1, 0, 0,  0,  0,  1, 0 ]
                  , [1, 2, 3, 4, 5, 6,  7,  8,  9, 10]) ≡
```

What is a Segmented Scan?

Assume an irregular matrix (two-dimensional array), i.e., rows have different number of elements.

Segmented Scan Intuition: the operation that scans each row of an irregular matrix and returns the result.

Flat Representation of a 2D irregular array: flag + value arrays:

```
[ [1, 2, 3]
, [4, 5, 6, 7, 8]
, [9, 10]
]
```

```
Flag Array:
[1, 0, 0, 1, 0, 0, 0, 0, 1, 0 ]
Value Array:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

The flag array marks with one (or something different than zero) the start of a row/segment, the other elements are zero.

The value array: flat sequence of elements.

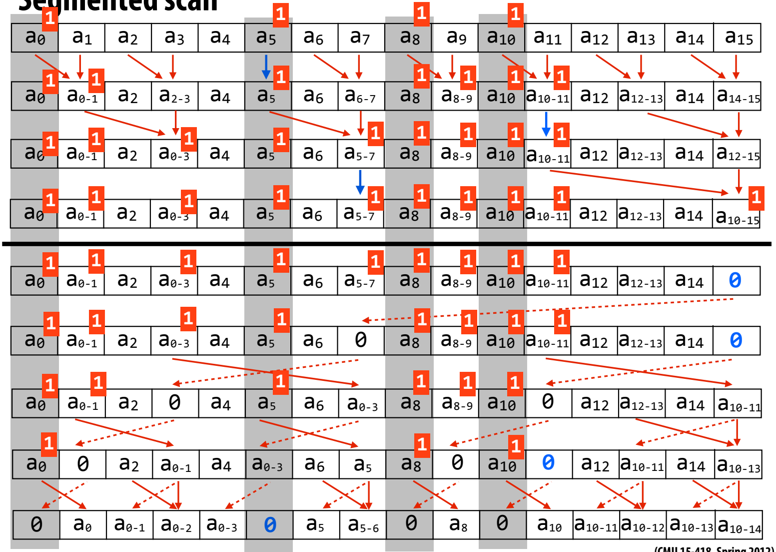
$\texttt{sgmScan}^{inc/exc}$: `((`$\alpha$`,`$\alpha$`)`$\to$ α`),` α`, [n]int, [n]`α`)` $\to$ `[n]`α

```
sgmScan^inc((+), 0, [1, 0, 0, 1, 0, 0,  0,  0,  1, 0 ]
                  , [1, 2, 3, 4, 5, 6,  7,  8,  9, 10]) ≡
                    [1, 3, 6, 4, 9, 15, 22, 30, 9, 19]
```

Intuition: How Does the Implementation Looks Like?

Slide taken from CMU 15-418: Parallel Computer Architecture and Programming (Spring 2012). Segmented Exclusive Scan:

Segmented scan



Segmented Exclusive Scan Algorithm And Complexity

- While there will be more branches, the asymptotic is the same as the one for reduce and scan:
- $D(n) = \Theta(lg\ n)$, $W(n) = \Theta(n)$!
- That's the good news!
- The bad news:

Segmented Exclusive Scan Algorithm And Complexity

- While there will be more branches, the asymptotic is the same as the one for reduce and scan:
- $D(n) = \Theta(lg\ n)$, $W(n) = \Theta(n)$!
- That's the good news!
- The bad news:

```
Input:   flag array F of n=2^k of ints
         data array A of n=2^k elems of type T
         ⊕ :: T × T → T associative
Output:  B = segmented scan of 2-dim array A
    FORALL i = 0 to n-1 do B[i] ← A[i] ENDDO
    FOR d = 0 to k-1 DO // up-sweep
      FORALL i = 0 to n-1 by 2^(d+1) DO
        IF F[i+2^(d+1)-1] == 0 THEN
            B[i+2^(d+1)-1] ← B[i+2^d-1] ⊕ B[i+2^(d+1)-1]
        ENDIF
        F[i+2^(d+1)-1] ← F[i+2^d-1] .|. F[i+2^(d+1)-1]
    ENDDO ENDDO
    B[n-1] ← 0
    FOR d = k-1 downto 0 DO // down-sweep
      FORALL i = 0 to n-1 by 2^(d+1) DO
        tmp ← B[i+2^d-1]
        IF F_original[i+2^d] ≠ 0 THEN
            B[i+2^(d+1)-1] ← 0
        ELSE IF F[i+2^d-1] ≠ 0 THEN
            B[i+2^(d+1)-1] ← tmp
        ELSE B[i+2^(d+1)-1] ← tmp ⊕ B[i+2^(d+1)-1]
        ENDIF
        F[i+2^(d+1)-1] ← 0
    ENDDO ENDDO
```

Segmented Inclusive Scan is a Sort of Scan!

Segmented Inclusive Scan is a Sort of Scan!

Can be implemented as a scan with a modified binary-associative operator that operates on boolean-value (or int-value) tuples.

We need to define `zip` first, which performs SoA to AoS transform:

zip: $([n]\alpha, [n]\beta) \rightarrow [n](\alpha, \beta)$

zip($[x_1, \ldots, x_n]$, $[y_1, \ldots, y_n]$) = $[(x_1, y_1), \ldots, (x_n, y_n)]$

Segmented Inclusive Scan is a Sort of Scan!

Can be implemented as a scan with a modified binary-associative operator that operates on boolean-value (or int-value) tuples.

We need to define `zip` first, which performs SoA to AoS transform:

```
zip:  ( [n]α, [n]β ) → [n](α,β)
zip([x₁,...,xₙ], [y₁,...,yₙ]) = [(x₁,y₁),...,(xₙ,yₙ)]

typedef Tuple {
    α    val;
    bool flg;
} Tup;

Tup ⊙^sgm(Tup x, Tup y) {
    Tup r;
    r.flg = x.flg || y.flg;
    r.val = y.flg ? y.val : x.val ⊙ y.val;
    return r;
}

α* sgmScan^inc (⊙ : (α,α) → α, α ne, α* flags, α* vals) {
    Tup nes; nes.flg = false; nes.val = ne;
    Tup* X = zip(flags, vals);
    return scan^inc(⊙^sgm, nes, X)
}
```

Sparse Matrix-Vector Multiplication: Pseudocode

```
smvMul( mat_cols: [N]int, mat_vals: [N]real
      , vct: [vct_len], shape: [num_rows]int ) : [num_rows]real =
  real res_vct[num_rows];
  int  offset = 0;
  for(int i=0; i < num_rows; i++) {
    real res = 0;
    for(int j=offset; j < offset+shape[i]; j++) {
        int vct_ind = mat_cols[j];
        int mat_elm = mat_vals[j];
        res += mat_elm * vct[vct_ind];
    }
    res_vct[i] = res;
    offset += shape[i];
  }

shape:    [2, 3, 1]
mat_cols: [0,    1,    1,   2,    3,   3  ]
mat_vals: [2.0, -1.0, -1.0, 2.0, -1.0, 3.0]
vct:      [1.0,  2.0,  3.0, 4.0]
-------------------------------------------
```

Sparse Matrix-Vector Multiplication: Pseudocode

```
smvMul( mat_cols: [N]int, mat_vals: [N]real
      , vct: [vct_len], shape: [num_rows]int ) : [num_rows]real =
  real res_vct[num_rows];
  int  offset = 0;
  for(int i=0; i < num_rows; i++) {
    real res = 0;
    for(int j=offset; j < offset+shape[i]; j++) {
        int vct_ind = mat_cols[j];
        int mat_elm = mat_vals[j];
        res += mat_elm * vct[vct_ind];
    }
    res_vct[i] = res;
    offset += shape[i];
  }
```

```
shape:    [2, 3, 1]
mat_cols: [0,   1,    1,   2,    3,   3  ]
mat_vals: [2.0, -1.0, -1.0, 2.0, -1.0, 3.0]
vct:      [1.0,  2.0,  3.0, 4.0]
-------------------------------------------
Result:
[ 2.0*1.0 + (-1.0)*2.0                 = 0.0
, (-1.0)*2.0 + 2.0*3.0 + (-1.0)*4.0    = 0.0
, 3.0*4.0                              = 12.0
]
```

Sparse Matrix-Vector Multiplication

Imperative Pseudocode:

```
smvMul( mat_cols: [N]int, mat_vals: [N]real
      , vct: [vct_len], shape: [num_rows]int ) : [num_rows]real =
  real res_vct[num_rows];
  int  offset = 0;
  for(int i=0; i < num_rows; i++) {
    real res = 0;
    for(int j=offset; j < offset+shape[i]; j++) {
        int vct_ind = mat_cols[j];
        int mat_elm = mat_vals[j];
        res += mat_elm * vct[vct_ind];
    }
    res_vct[i] = res;
    offset += shape[i];
  }
```

We would like to:

- exploit all parallelism, but the inner parallelism is irregular!
- the inner loop:

Sparse Matrix-Vector Multiplication

Imperative Pseudocode:

```
smvMul( mat_cols: [N]int, mat_vals: [N]real
      , vct: [vct_len], shape: [num_rows]int ) : [num_rows]real =
  real res_vct[num_rows];
  int  offset = 0;
  for(int i=0; i < num_rows; i++) {
    real res = 0;
    for(int j=offset; j < offset+shape[i]; j++) {
        int vct_ind = mat_cols[j];
        int mat_elm = mat_vals[j];
        res += mat_elm * vct[vct_ind];
    }
    res_vct[i] = res;
    offset += shape[i];
  }
```

We would like to:

- exploit all parallelism, but the inner parallelism is irregular!
- the inner loop:looks like a map-reduce composition,
  buts its size is variant to the outer map;
- Hint: compute flags + use segmented scan + extract last elements!

Computing Flag Array from Shape Array

This works even in the presence of empty rows!

```
mkFlagArray (aoa_shp: [m]i32) : []i32 =   aoa_shp=[0,3,1,0,4,2,0]
  shp_rot = map (λi->if i==0 then 0
                     else aoa_shp[i-1]
                , [0,...,m-1]);
  shp_scn = scan( (+), 0, shp_rot );
  aoa_len = shp_scn[m-1] + aoa_shp[m-1];
  shp_ind = map2 (λshp ind ->
                   if shp==0 then -1
                            else ind
                 , aoa_shp, shp_scn);
  return
    scatter( replicate(aoa_len, 0)
           , shp_ind
           , replicate(m, 1) );
```

Computing Flag Array from Shape Array

This works even in the presence of empty rows!

```
mkFlagArray (aoa_shp: [m]i32) : []i32 =   aoa_shp=[0,3,1,0,4,2,0]
  shp_rot = map (λi->if i==0 then 0       shp_rot=[0,0,3,1,0,4,2]
                     else aoa_shp[i-1]
                , [0,...,m-1]);
  shp_scn = scan( (+), 0, shp_rot );
  aoa_len = shp_scn[m-1] + aoa_shp[m-1];
  shp_ind = map2 (λshp ind ->
                  if shp==0 then -1
                            else ind
                 , aoa_shp, shp_scn);
  return
    scatter( replicate(aoa_len, 0)
           , shp_ind
           , replicate(m, 1) );
```

Computing Flag Array from Shape Array

This works even in the presence of empty rows!

```
mkFlagArray (aoa_shp: [m]i32) : []i32 =
  shp_rot = map (λi->if i==0 then 0
                     else aoa_shp[i-1]
                , [0,...,m-1]);
  shp_scn = scan( (+), 0, shp_rot );
  aoa_len = shp_scn[m-1] + aoa_shp[m-1];
  shp_ind = map2 (λshp ind ->
                   if shp==0 then -1
                             else ind
                 , aoa_shp, shp_scn);
  return
    scatter( replicate(aoa_len, 0)
           , shp_ind
           , replicate(m, 1) );
```

```
aoa_shp=[0,3,1,0,4,2,0]
shp_rot=[0,0,3,1,0,4,2]

shp_scn=[0,0,3,4,4,8,10]
```

Computing Flag Array from Shape Array

This works even in the presence of empty rows!

```
mkFlagArray (aoa_shp: [m]i32) : []i32 =
  shp_rot = map (λi->if i==0 then 0
                     else aoa_shp[i-1]
                , [0,...,m-1]);
  shp_scn = scan( (+), 0, shp_rot );
  aoa_len = shp_scn[m-1] + aoa_shp[m-1];
  shp_ind = map2 (λshp ind ->
                 if shp==0 then -1
                           else ind
                 , aoa_shp, shp_scn);
  return
    scatter( replicate(aoa_len, 0)
           , shp_ind
           , replicate(m, 1) );
```

```
aoa_shp=[0,3,1,0,4,2,0]
shp_rot=[0,0,3,1,0,4,2]

shp_scn=[0,0,3,4,4,8,10]
aoa_len= 10

shp_ind=
```

Computing Flag Array from Shape Array

This works even in the presence of empty rows!

```
mkFlagArray (aoa_shp: [m]i32) : []i32 =
  shp_rot = map (λi->if i==0 then 0
                     else aoa_shp[i-1]
                , [0,...,m-1]);
  shp_scn = scan( (+), 0, shp_rot );
  aoa_len = shp_scn[m-1] + aoa_shp[m-1];
  shp_ind = map2 (λshp ind ->
                   if shp==0 then -1
                             else ind
                 , aoa_shp, shp_scn);
  return
    scatter( replicate(aoa_len, 0)
           , shp_ind
           , replicate(m, 1) );
```

```
aoa_shp=[0,3,1,0,4,2,0]
shp_rot=[0,0,3,1,0,4,2]

shp_scn=[0,0,3,4,4,8,10]
aoa_len= 10

shp_ind=
      [-1,0,3,-1,4,8,-1]
scatter
```

Computing Flag Array from Shape Array

This works even in the presence of empty rows!

```
mkFlagArray (aoa_shp: [m]i32) : []i32 =   aoa_shp=[0,3,1,0,4,2,0]
  shp_rot = map (λi->if i==0 then 0       shp_rot=[0,0,3,1,0,4,2]
                     else aoa_shp[i-1]
                 , [0,...,m-1]);           shp_scn=[0,0,3,4,4,8,10]
  shp_scn = scan( (+), 0, shp_rot );      aoa_len= 10
  aoa_len = shp_scn[m-1] + aoa_shp[m-1];
  shp_ind = map2 (λshp ind ->             shp_ind=
                  if shp==0 then -1              [-1,0,3,-1,4,8,-1]
                           else ind       scatter
                 , aoa_shp, shp_scn);       [0,0,0,0,0,0,0,0,0,0]
  return                                    [-1,0,3,-1,4,8,-1]
    scatter( replicate(aoa_len, 0)          [1,1,1,1,1,1,1]
           , shp_ind                        --------------------
           , replicate(m, 1) );             [1,0,0,1,1,0,0,0,1,0]
```

Flat-Parallel Sparse Matrix-Vector Mult

Idea: use segmented scan, then extract last element in every row!

We assume no rows of length 0, and also inline flag computation.

```
smvMul( mat_cols: [N]int        // [0,    1,    1,   2,    3,   3  ]
      , mat_vals: [N]real       // [2.0, -1.0, -1.0, 2.0, -1.0, 3.0]
      , vct: [vct_len]          // [1.0,  2.0,  3.0, 4.0]
      , shape: [num_rows]int )  // [2,    3,    1]
      : [num_rows]real =        // Expected: [0.0, 0.0, 12.0]
  shpscn= scan( (+), 0, shape );        // [2, 5, 6]
  len   = shpscn[num_rows-1];
  ones  = replicate(num_rows, 1);       // [1, 1, 1]
  indm1 = map(λ i -> if i==0 then 0
                     else shpscn[i-1]
                    , [0...num_rows]); // [0, 2, 5]
  flags = scatter( replicate(len, 0)   // [0, 0, 0, 0, 0, 0]
                 , indm1
                 , ones );      // [1,    0,    1,   0,    0,   1  ]
  prods = map2(λcol v ->        // [2.0, -2.0, -2.0, 6.0, -4.0, 12.0]
                    v * vct[col]
              , mat_cols, mat_vals );
  scnarr= sgmScan^inc( (+), 0
              , flags, prods); // [2.0, 0.0, -2.0, 4.0, 0.0, 12.0]
```

Flat-Parallel Sparse Matrix-Vector Mult

Idea: use segmented scan, then extract last element in every row!

We assume no rows of length 0, and also inline flag computation.

```
smvMul( mat_cols: [N]int          // [0,    1,    1,   2,    3,   3  ]
      , mat_vals: [N]real         // [2.0, -1.0, -1.0, 2.0, -1.0, 3.0]
      , vct: [vct_len]            // [1.0,  2.0,  3.0, 4.0]
      , shape: [num_rows]int )    // [2,    3,    1]
      : [num_rows]real =          // Expected: [0.0, 0.0, 12.0]
  shpscn= scan( (+), 0, shape );          // [2, 5, 6]
  len   = shpscn[num_rows-1];
  ones  = replicate(num_rows, 1);         // [1, 1, 1]
  indm1 = map(λ i -> if i==0 then 0
                     else shpscn[i-1]
                    , [0...num_rows]);    // [0, 2, 5]
  flags = scatter( replicate(len, 0)      // [0, 0, 0, 0, 0, 0]
                 , indm1
                 , ones );        // [1,    0,    1,   0,    0,   1  ]
  prods = map2(λcol v ->          // [2.0, -2.0, -2.0, 6.0, -4.0, 12.0]
                    v * vct[col]
              , mat_cols, mat_vals );
  scnarr= sgmScan^inc( (+), 0
              , flags, prods);    // [2.0, 0.0, -2.0, 4.0, 0.0, 12.0]
  return                          // [scnarr[1], scn_arr[4], scn_arr[5]]
    map(λ ind -> scnarr[ind-1];
       , shpscn );                // [0.0, 0.0, 12.0]
```

Exercise 4: Sparse Matrix-Vector Mult in OpenCL

- the CPU stub is in `Day3-Exercises/ScanApps/spMatVecMult.h`, function `runSpMatVectMul`; nothing to do there! You only need to implement several kernels in `scanapps.cl`.

(1) `shpscn= scan( (+), 0, shape );`
is already implemented!

(2) Implement kernel `mkFlagsSpMVM`. Array `flags` has been already zeroed; `shape_scn` is `shpscn` in the previous slide.

(3) Implement kernel `mulPhaseSpMVM`.
`mat_ind` is `mat_cols` from the previous slide.

(4) `scnarr=` $\texttt{sgmScan}^{inc}$`( (+), 0, flags, prods);`
is already implemented!

(5) Implement kernel `getLastSpMVM`.
`shape_scn` and `sgm_mat` are `shpscn` and `scnarr` in previous slide, respectively, and `out` is the result.

Course Contents

Back To Partition2

```
partition2(X: [n]i32): (i32, [n]i32) =
 cs = map(even, X);
 tfs= map (λc->if c then 1 else 0
          , cs);
 isT= scan( (+), 0, tfs );
 i  = isT[n-1];

 ffs= map (λc->if c then 0 else 1
          , cs );
 tmps = scan( (+), 0, ffs );
 isF= map (λt -> t+i, tmps);

 inds=map3(λ(c,iT,iF) ->
             if c then iT-1 else iF-1
          , cs, isT, isF);
 res = scatter(scratch n, inds, X);
 return (i, res);
```

```
X    = [5, 4, 2, 10,3, 7, 8]
n    = 7
cs   = [F, T, T, T, F, F, T]
tfs  = [0, 1, 1, 1, 0, 0, 1]

isT  = [0, 1, 2, 3, 3, 3, 4]
i    = 4

ffs  = [1, 0, 0, 0, 1, 1, 0]
tmps= [1, 1, 1, 1, 2, 3, 3]
isF  = [5, 5, 5, 5, 6, 7, 7]

inds= [4, 0, 1, 2, 5, 6, 3]
X    = [5, 4, 2, 10,3, 7, 8]
Res  = [_, _, _, _, _, _, _]
-----------------------------
Res  = [4, 2, 10, 8,5, 3, 7]
```

This shows you how to think about it, but a direct implementation is not very efficient in practice.

Partition2: Taking Inspiration From Scan

```
bool even(int v) { return (bool)(1 -(v&1)); }
partition2(even, [5, 4, 2, 10, 3, 7, 8] ) should result in
             (4, [4, 2, 10, 8, 5, 3, 7])
```

A scan implementation is typically split into three kernels:

(1) reduce at workgroup level (with sequential chunking);

(2) scan the results of (1) in one workgroup; this is typically negligible, as it is applied on few elements;

(3) perform the workgroup level scan while starting from the result of the previous workgroup.

Partition2: Deriving an Optimized Implementation

```
bool even(int v) { return (bool)(1 -(v&1)); }
partition2(even, [5, 4, 2, 10, 3, 7, 8] ) should result in
             (4, [4, 2, 10, 8, 5, 3, 7])
```

(I) map-reduce at workgroup level; operator is commutative!
   (a) `X = map(λ(e)-> if pred(e) then (1,0) else (0,1), A);`
   (b) reduce^{wkgp}`(λ((a1,b1),(a2,b2))->(a1+b2, b1+b2)),(0,0),X);`

(II) scan the results of (I) in one workgroup; after this step we know the global offsets of the elements that succeed/fail at the start of each workgroup.

(III) Do all the rest in one kernel:
   (1) Copy ELEMS_PER_THREAD from global to local memory, coalesced (ToDo).
   (2) Save them in per-thread register memory (chunk) +
   Reduce sequentially each thread's ELEMS_PER_THREAD elems in `tf`; (ToDo).
   (3) Perform workgroup-level scan on the `tf` of each thread (DONE).
   (4) The number of elements that succeed/fail per workgroup (`num_ts`/`num_fs`) is the last scanned element (DONE).
   (5) Save in `tf` the result of the previous thread (DONE).
   (6) Redo computation (2) but instead place the elements from register to local memory in the order in which they would occur if the partition was done at workgroup level; remember to add `num_ts` to the failed-elems index (ToDo).
   (7) Finally copy-out from local to global memory in coalesced fashion (DONE).

Last Exercise: Optimized Partition2

- CPU stub: in file `Day3-Exercises/Partition2Opt/partition2.h`, function `runPartition`; that is where the OpenCL kernels are called, but nothing to do there.

- Open in file `Day3-Exercises/Partition2Opt/kernels.cl`, kernel `scanPhaseKer`. Follow the instructions, which correspond to implementing the ToDo items from the previous slide.

- The performance of the optimized partition2 should come close to that of inclusive scan!