Futhark

Purely Functional GPU-Programming with Nested Parallelism and In-Place Array Updates

Troels Henriksen
Niels G. W. Serup
Martin Elsman
Fritz Henglein
Cosmin Oancea

DIKU
University of Copenhagen

20th of June 2017

The Problem

Problem: Modern massively parallel machines (e.g. GPUs) require specialist expertise to obtain good performance, and sequential languages are a bad fit.

The Problem

Problem: Modern massively parallel machines (e.g. GPUs) require specialist expertise to obtain good performance, and sequential languages are a bad fit.

Solution: Use a purely functional language with data-parallel operators!

The New Problem

Problem: Turns out purely functional languages are really slow when compiled naively, and GPUs only support certain restricted forms of parallelism anyway.

The New Problem

Problem: Turns out purely functional languages are really slow when compiled naively, and GPUs only support certain restricted forms of parallelism anyway.

Solution: Spend four years co-designing a simple language and a non-simple optimising compiler capable of compiling it to efficient GPU code: **Futhark!**

Futhark at a Glance

Small eagerly evaluated pure functional language with data-parallel looping constructs. Syntax is a combination of C, SML, and Haskell.

- **Data-parallel loops**

```
let add_two (a:  [n]i32):     [n]i32 = map (+2) a
let     sum (a:  [n]i32):        i32 = reduce (+) 0 a
let sumrows (as: [n][m]i32): [n]i32 = map sum as
```

Futhark at a Glance

- **Array construction**

```
iota 5             = [0,1,2,3,4]
replicate 3 1337 = [1337, 1337, 1337]
```

  Only regular arrays: `[[1,2], [3]]` is illegal.

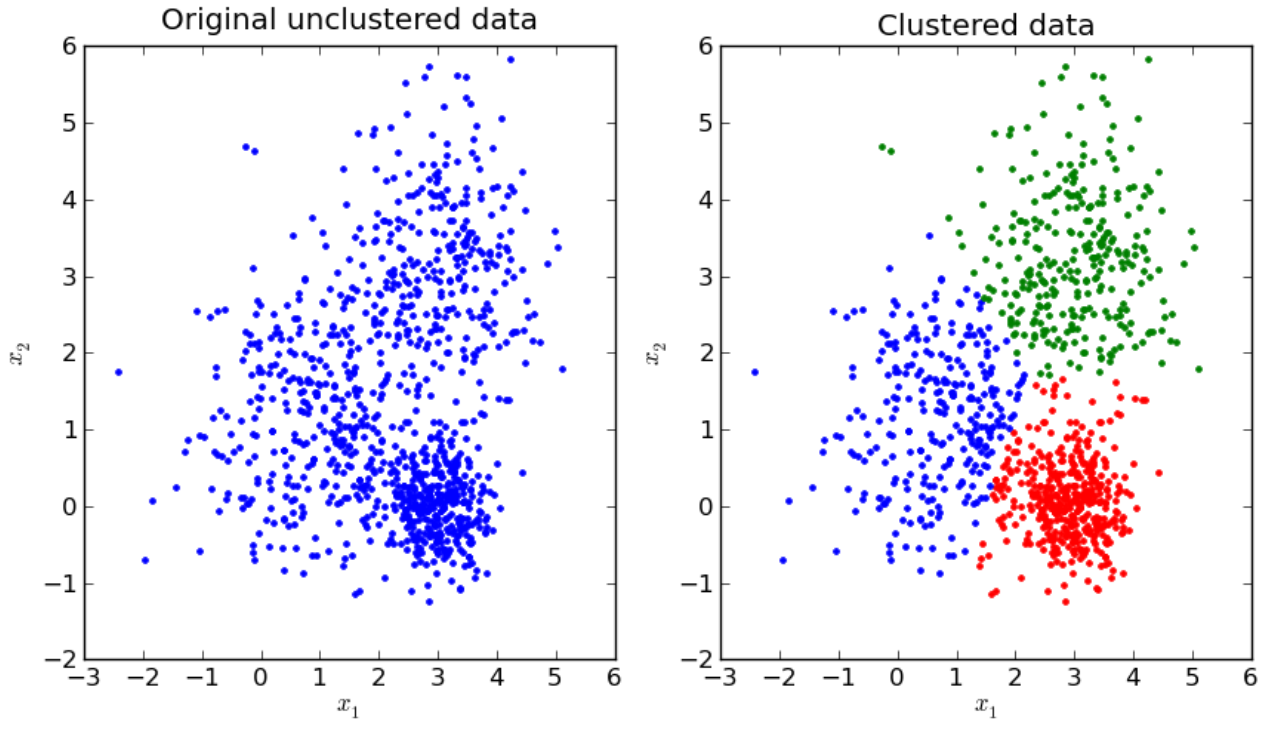
- **Sequential loops**

```
loop (x = 1) for i < n do
  x * (i + 1)
```

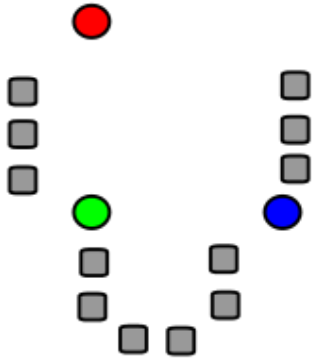
Case Study:
k-means Clustering

The Problem

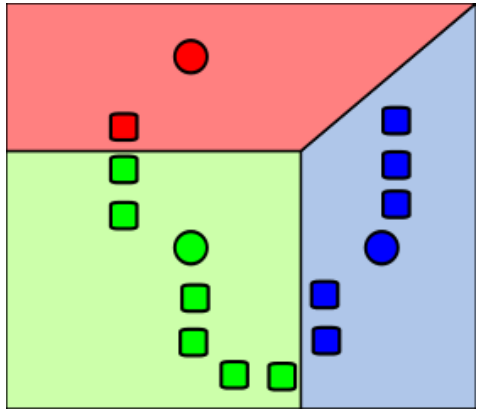
We are given n points in some d-dimensional space, which we must partition into k disjoint sets, such that we minimise the inter-cluster sum of squares (the distance from every point in a cluster to the centre of the cluster).
Example with $d = 2, k = 3, n =$ *more than I can count*:



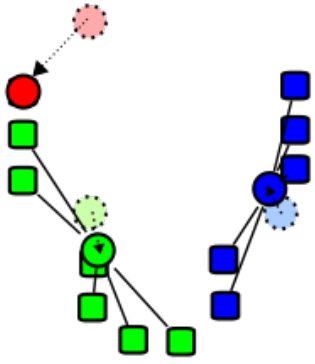
The Solution (from Wikipedia)



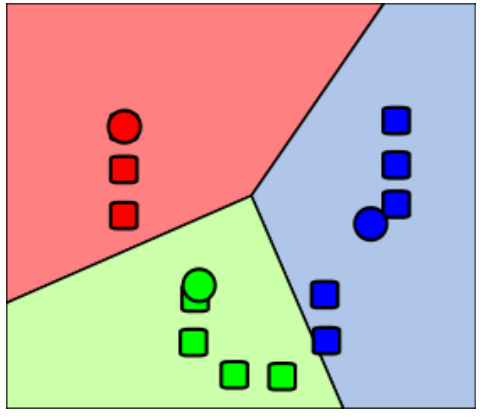
(1) k initial "means" (here $k = 3$) are randomly generated within the data domain.



(2) k clusters are created by associating every observation with the nearest mean.

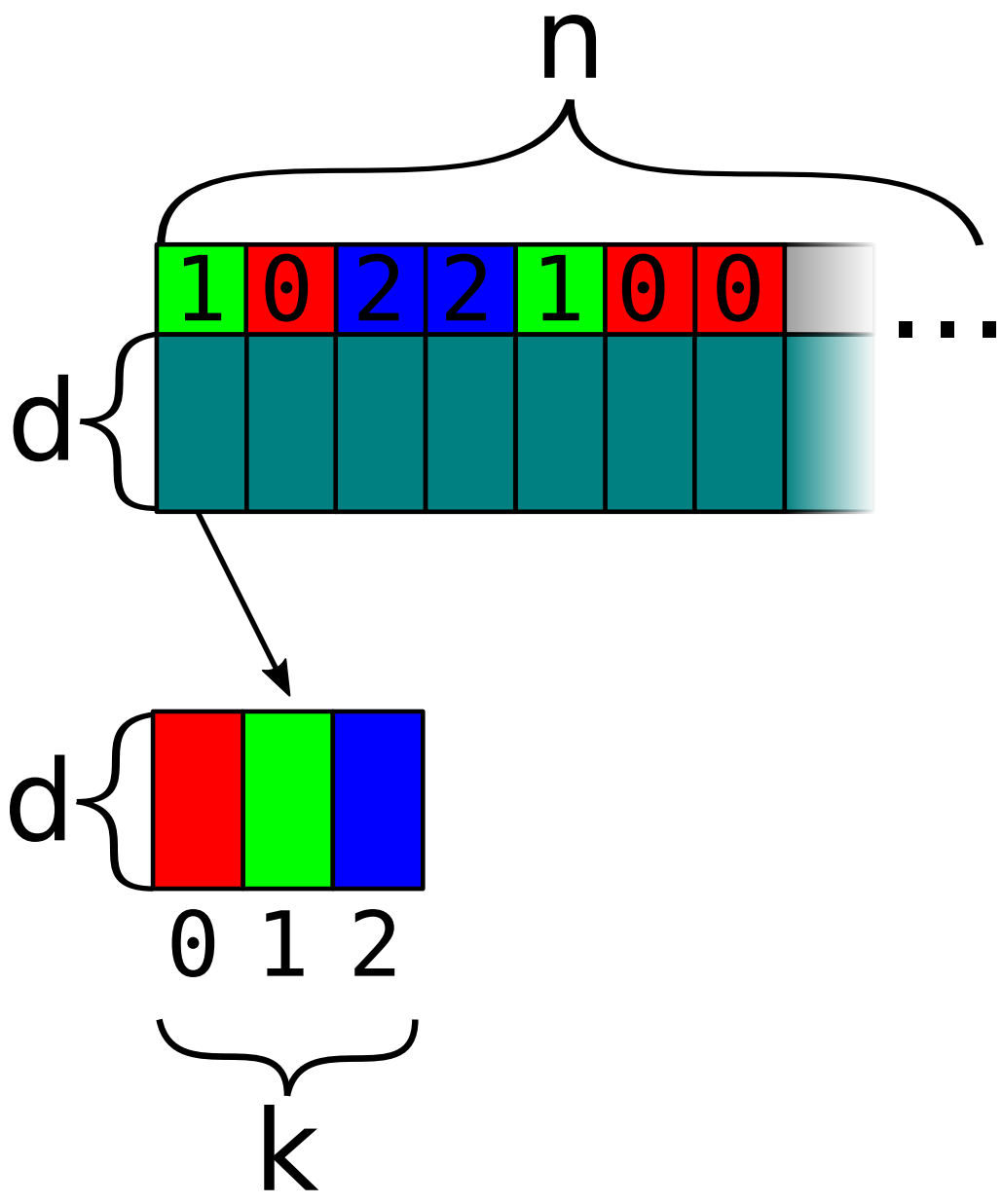


(3) The centroid of each of the k clusters becomes the new mean.

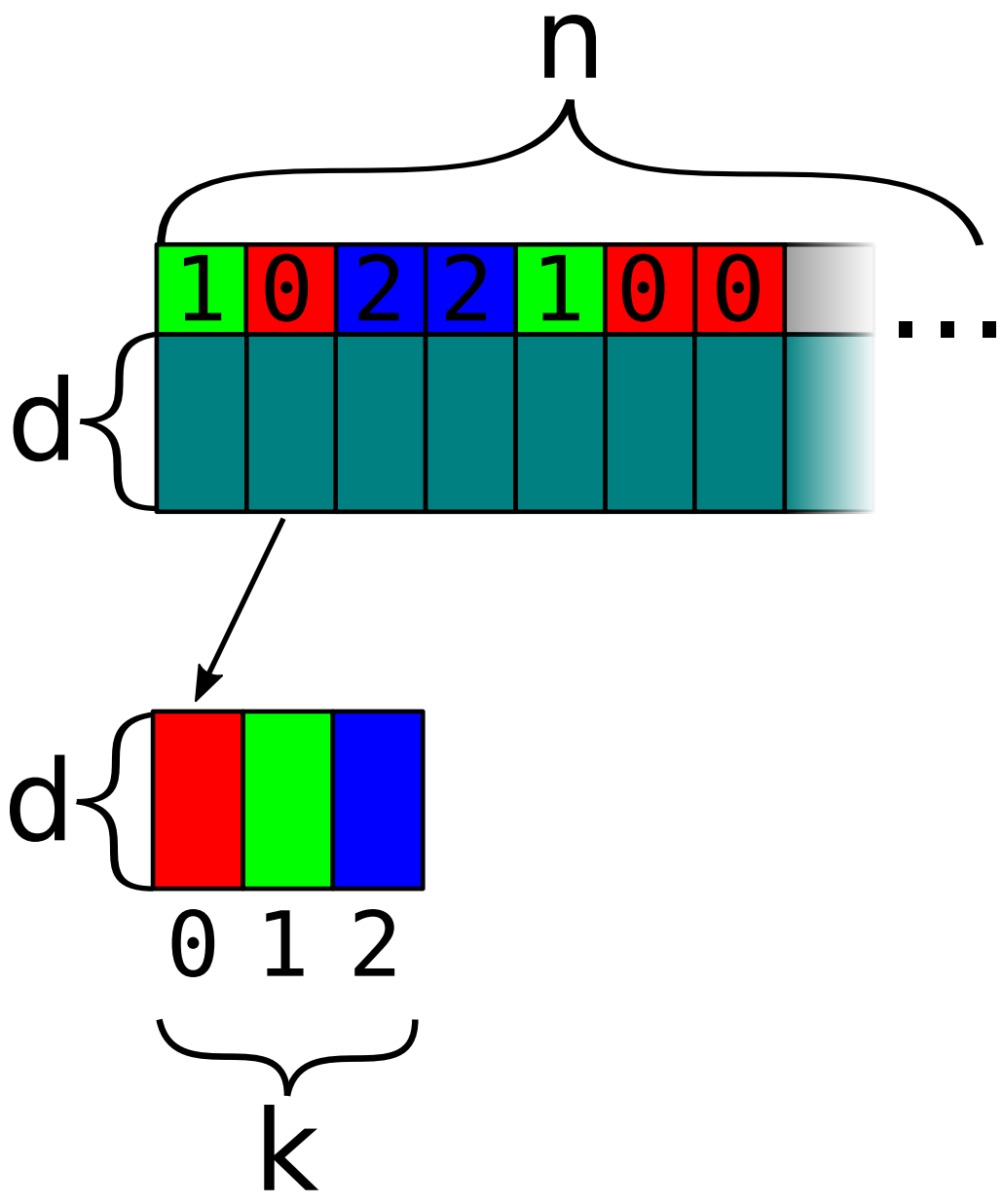


(4) Steps (2) and (3) are repeated until convergence has been reached.

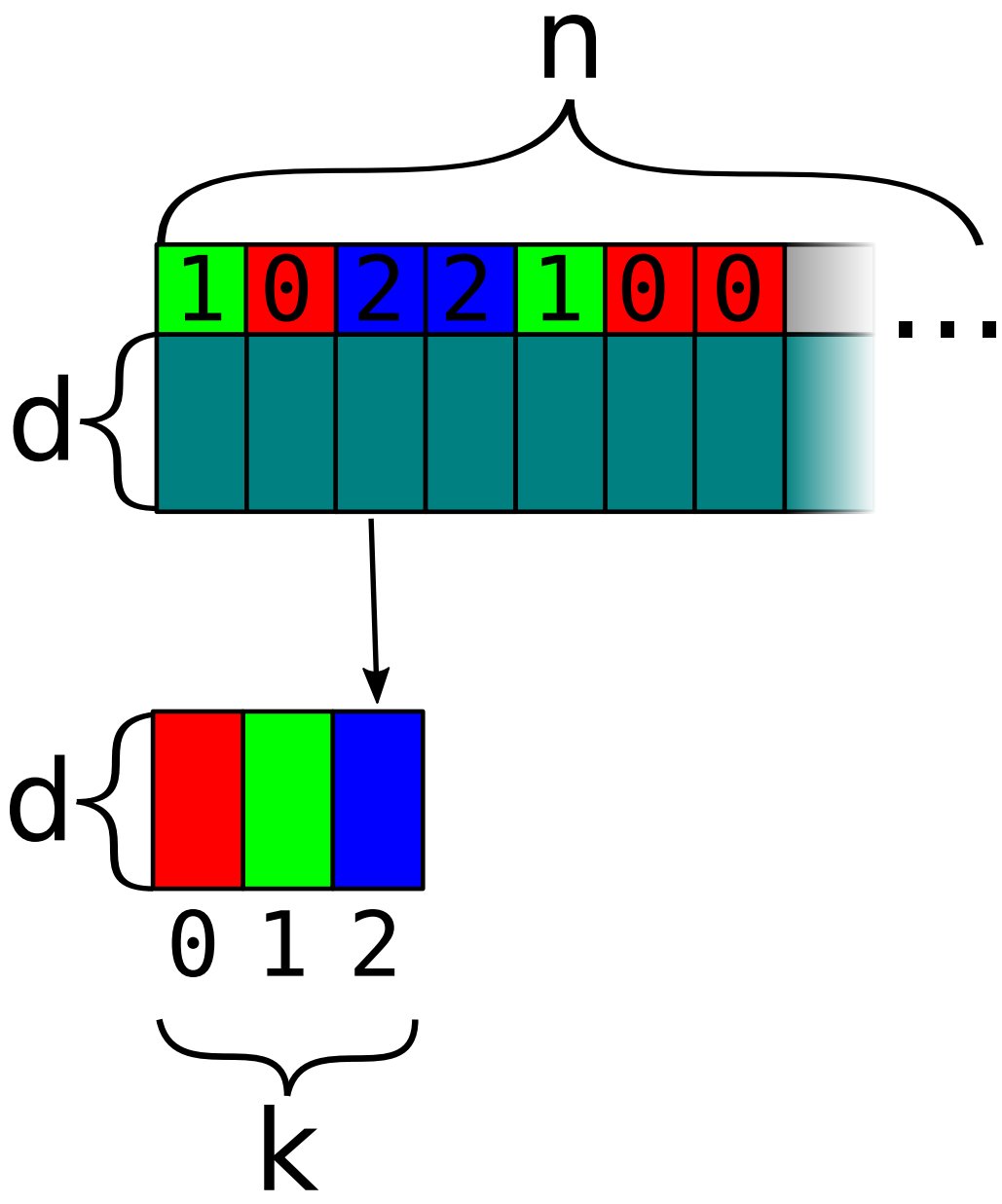
Computing Cluster Means: Fully Sequential



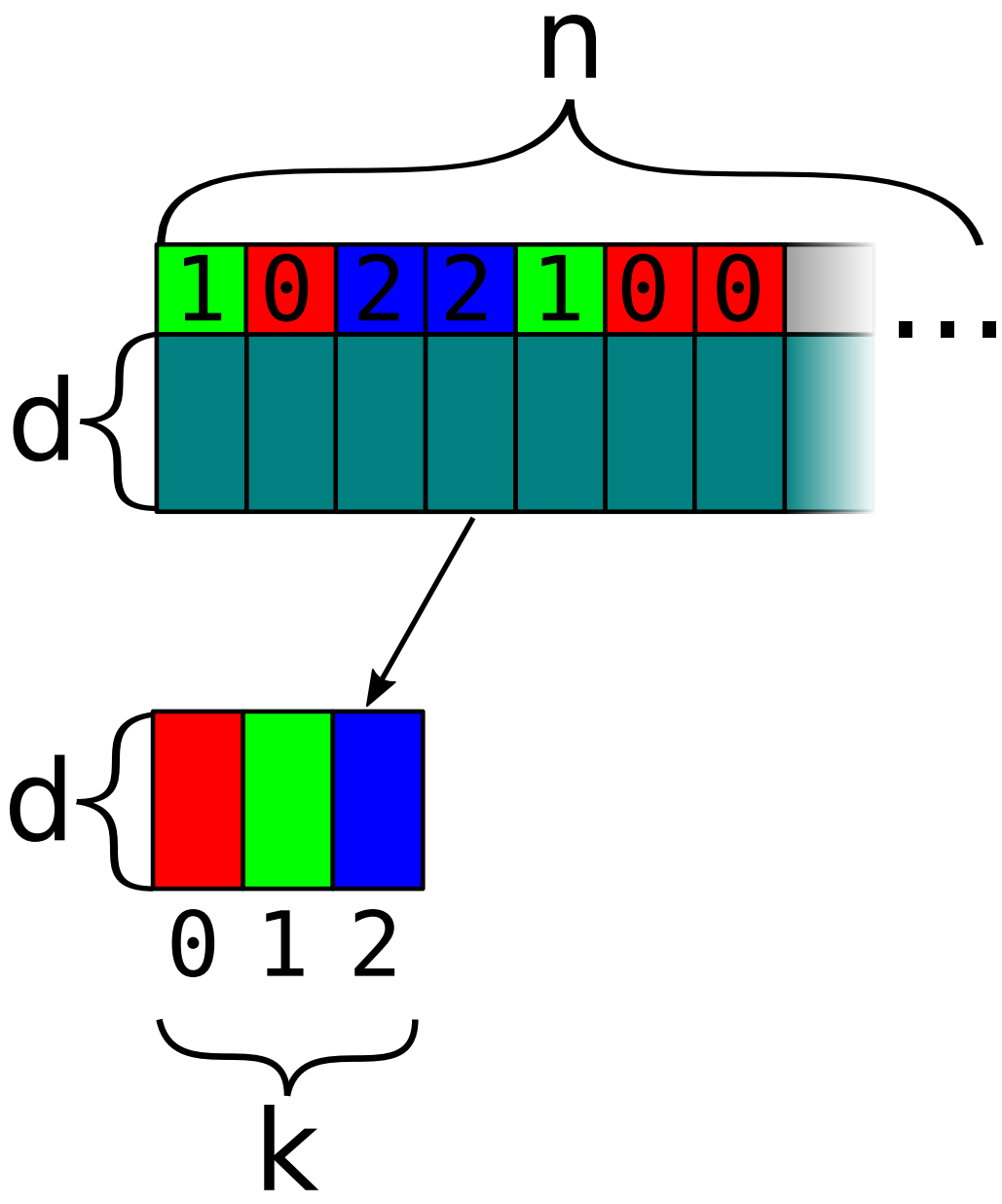
Computing Cluster Means: Fully Sequential



Computing Cluster Means: Fully Sequential



Computing Cluster Means: Fully Sequential



Computing Cluster Means: Fully Sequential

```
let add_points(x: [d]f32) (y: [d]f32): [d]f32 =
  map (+) x y

let cluster_means_seq (cluster_sizes: [k]i32)
                      (points: [n][d]f32)
                      (membership: [n]i32): [k][d]f32 =
  loop (acc = replicate k (replicate d 0.0)) for i < n do
    let p = points[i]
    let c = membership[i]
    let p' = map (/f32(cluster_sizes[c])) p
    in acc with [c] ← add_points acc[c] p'
```

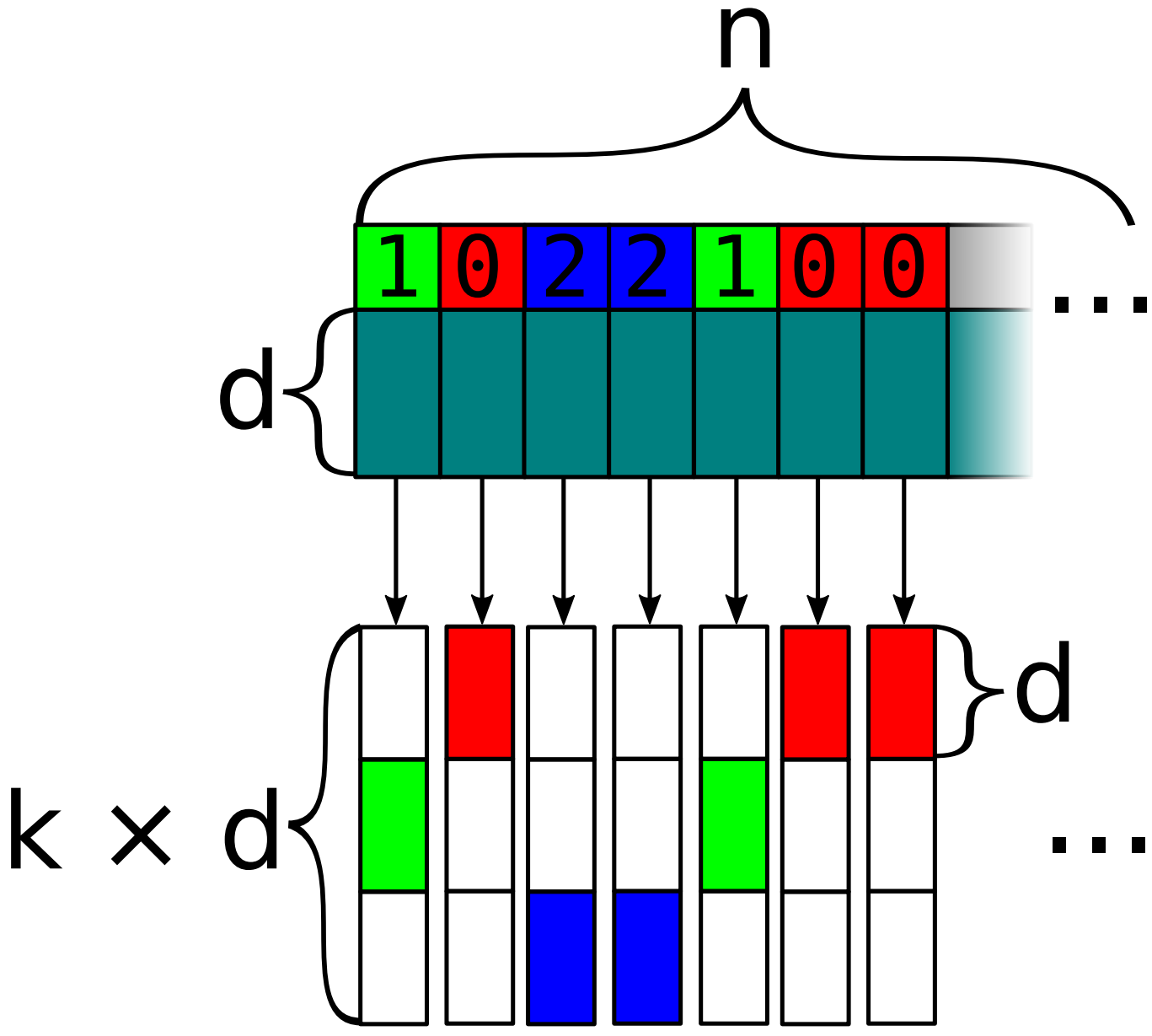
Computing Cluster Means: Fully Sequential

```
let add_points(x: [d]f32) (y: [d]f32): [d]f32 =
  map (+) x y

let cluster_means_seq (cluster_sizes: [k]i32)
                      (points: [n][d]f32)
                      (membership: [n]i32): [k][d]f32 =
  loop (acc = replicate k (replicate d 0.0)) for i < n do
    let p = points[i]
    let c = membership[i]
    let p' = map (/f32(cluster_sizes[c])) p
    in acc with [c] ← add_points acc[c] p'
```

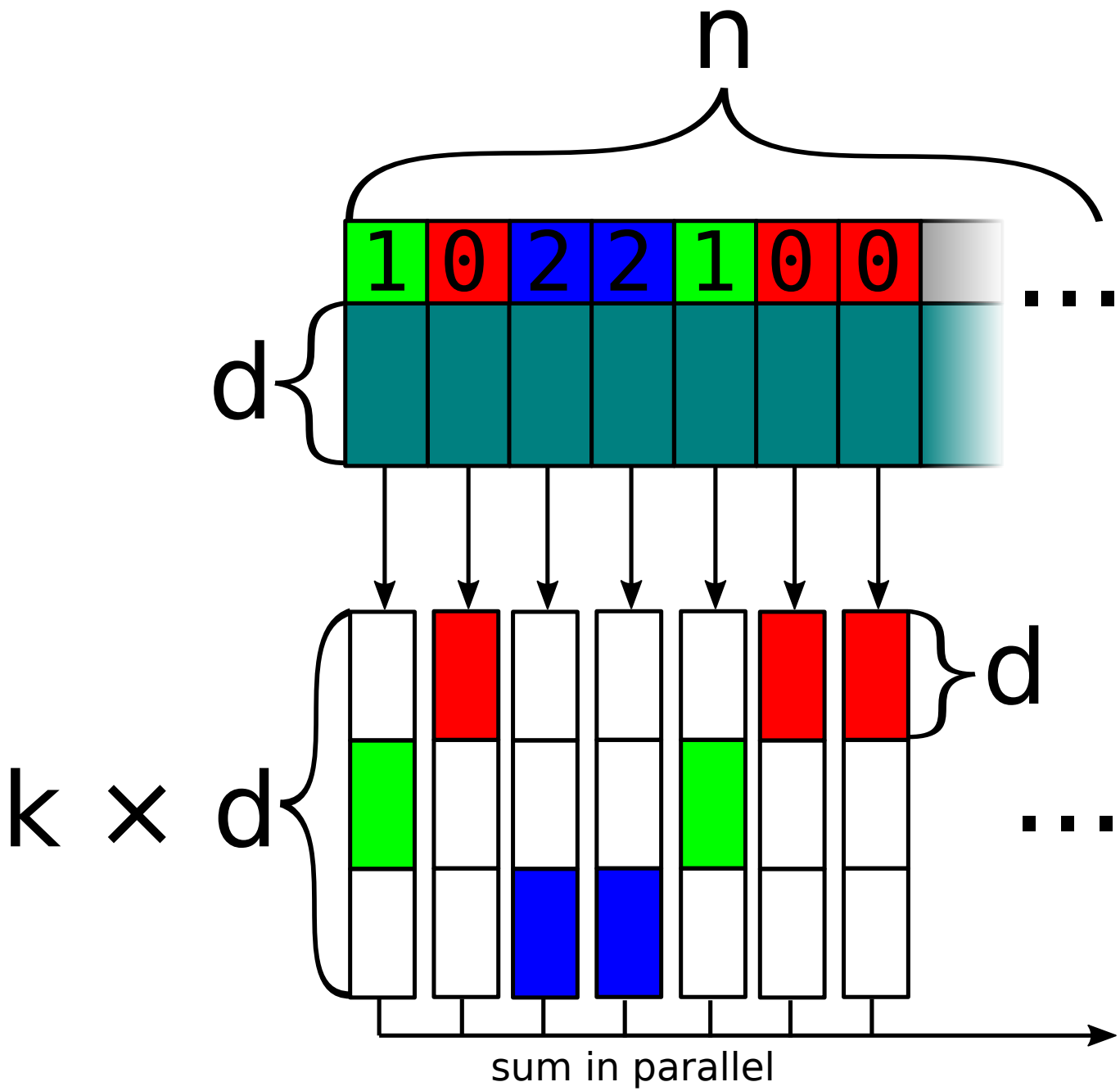
Problem

$O(n \times d)$ work, but not much parallelism.

Computing Cluster Means: Fully Parallel



Computing Cluster Means: Fully Parallel



Computing Cluster Means: Fully Parallel

Use a parallel `map` to compute "increments", and then a `reduce` of these increments.

```
let matrix_add (xss: [k][d]f32) (yss: [k][d]f32): [k][d]f32 =
  map (λxs ys → map (+) xs ys) xss yss

let cluster_means_par(cluster_sizes: [k]i32)
                     (points: [n][d]f32)
                     (membership: [n]i32): [k][d]f32 =
  let increments : [n][k][d]i32 =
    map (λp c →
           let a = replicate k (replicate d 0.0)
           let p' = map (/(f32(cluster_sizes[c]))) p
           in a with [c] ← p')
        points membership
  in reduce matrix_add (replicate k (replicate d 0.0))
            increments
```

Computing Cluster Means: Fully Parallel

Use a parallel `map` to compute "increments", and then a `reduce` of these increments.

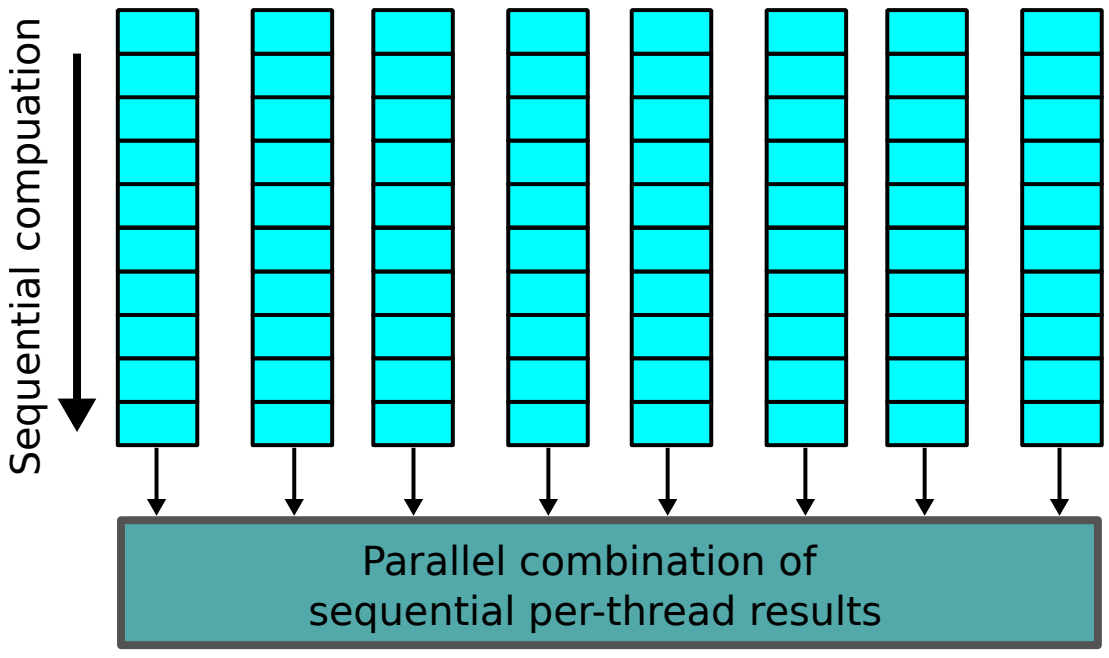
```
let matrix_add (xss: [k][d]f32) (yss: [k][d]f32): [k][d]f32 =
  map (λxs ys → map (+) xs ys) xss yss

let cluster_means_par(cluster_sizes: [k]i32)
                     (points: [n][d]f32)
                     (membership: [n]i32): [k][d]f32 =
  let increments : [n][k][d]i32 =
    map (λp c →
           let a = replicate k (replicate d 0.0)
           let p' = map (/(f32(cluster_sizes[c]))) p
           in a with [c] ← p')
        points membership
  in reduce matrix_add (replicate k (replicate d 0.0))
            increments
```

Problem

Fully parallel, but $O(k \times n \times d)$ work.

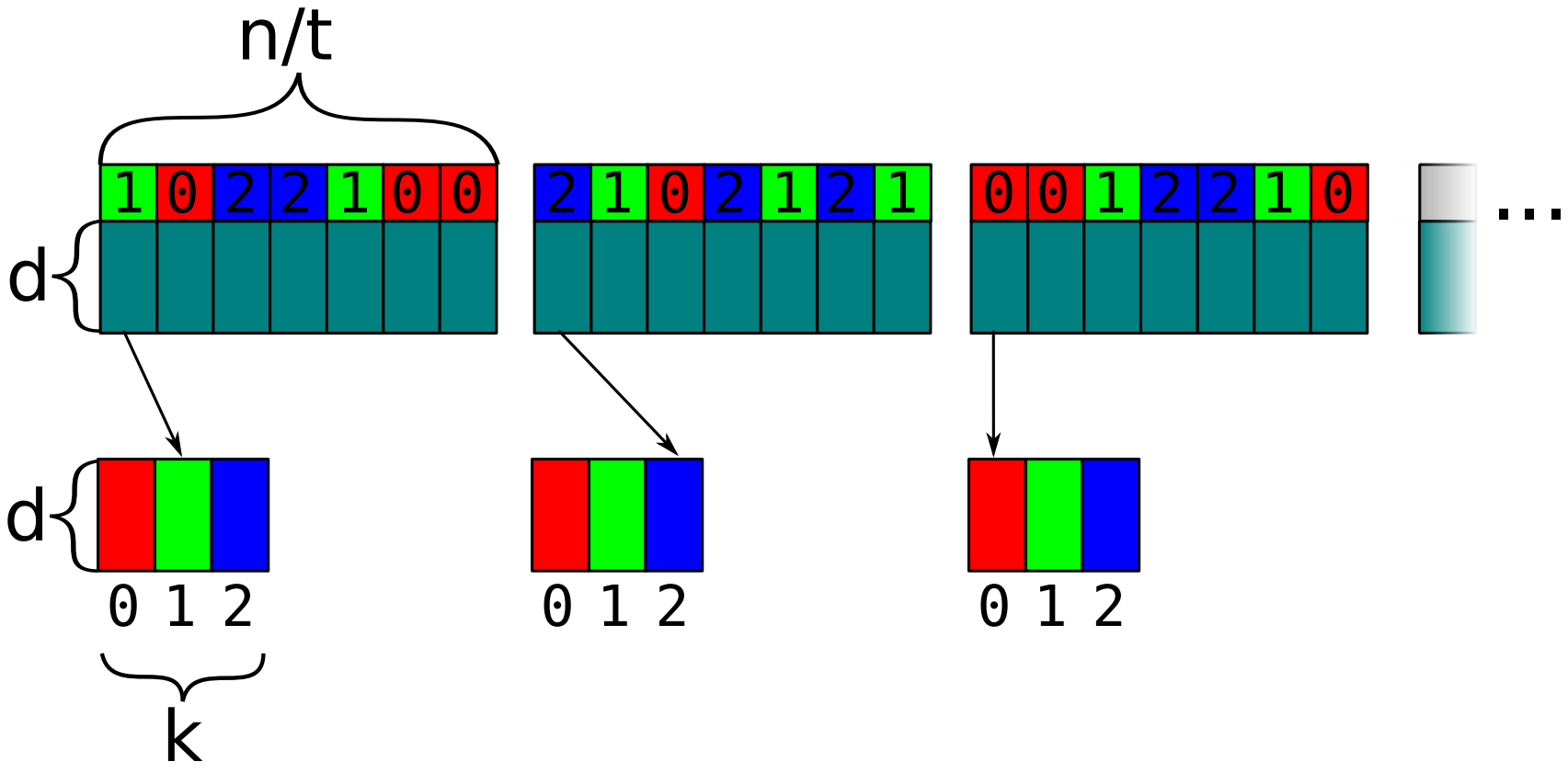
One Futhark Design Principle

The hardware is not infinitely parallel - ideally, we use an efficient sequential algorithm for chunks of the input, then use a parallel operation to combine the results of the sequential parts.

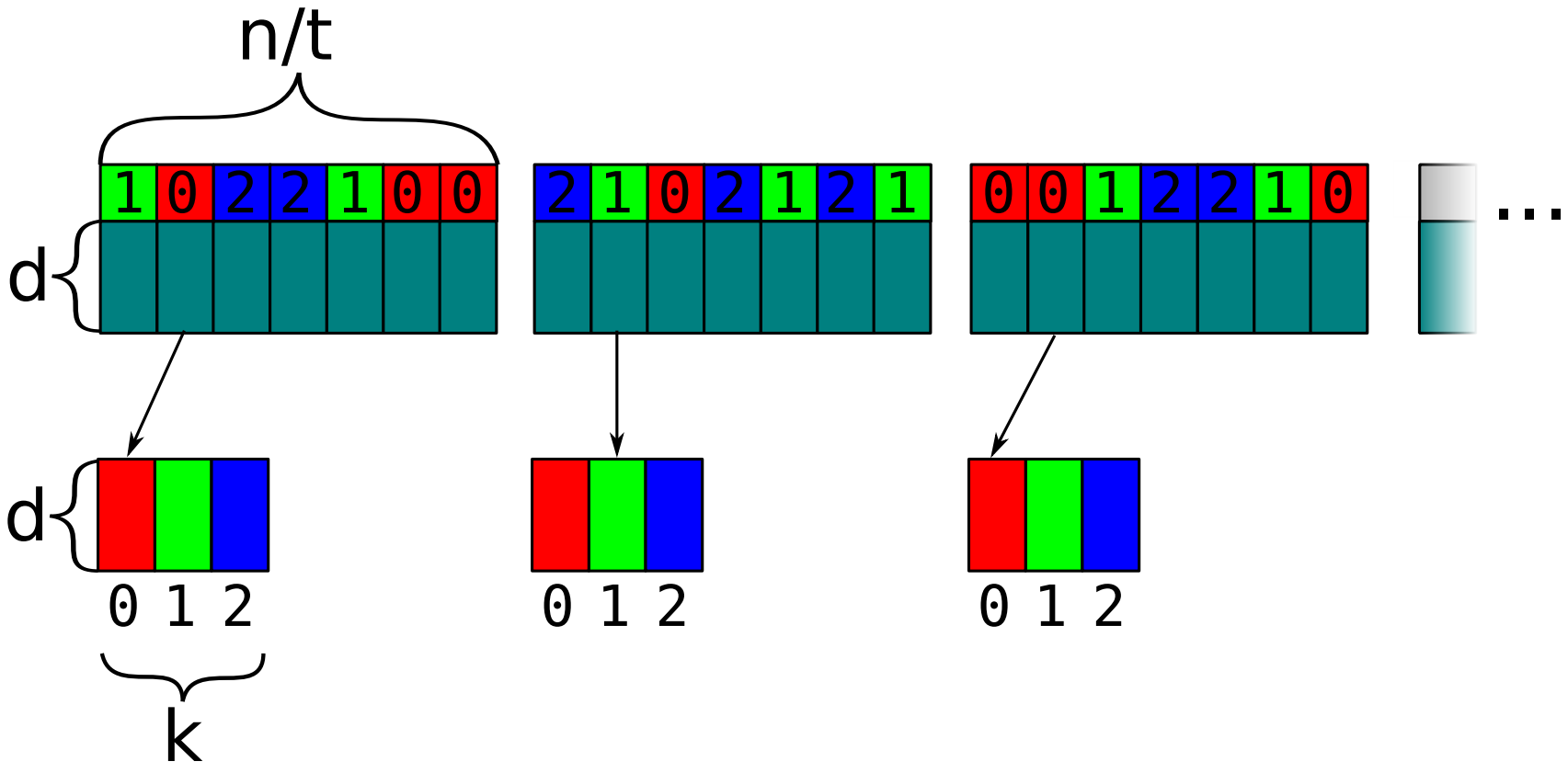


The optimal number of threads varies from case to case, so this should be abstracted from the programmer.

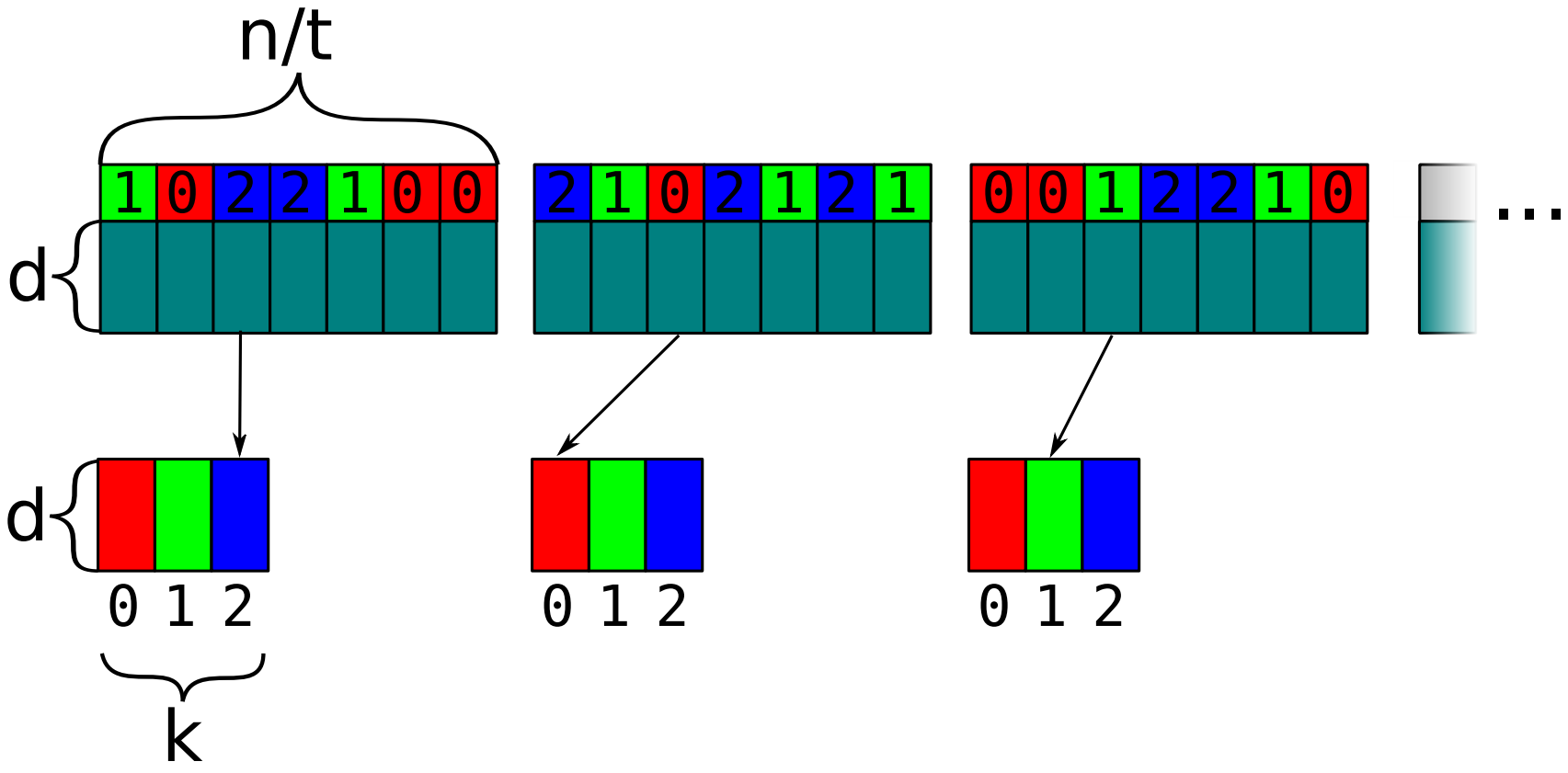
Computing Cluster Means: Just Right



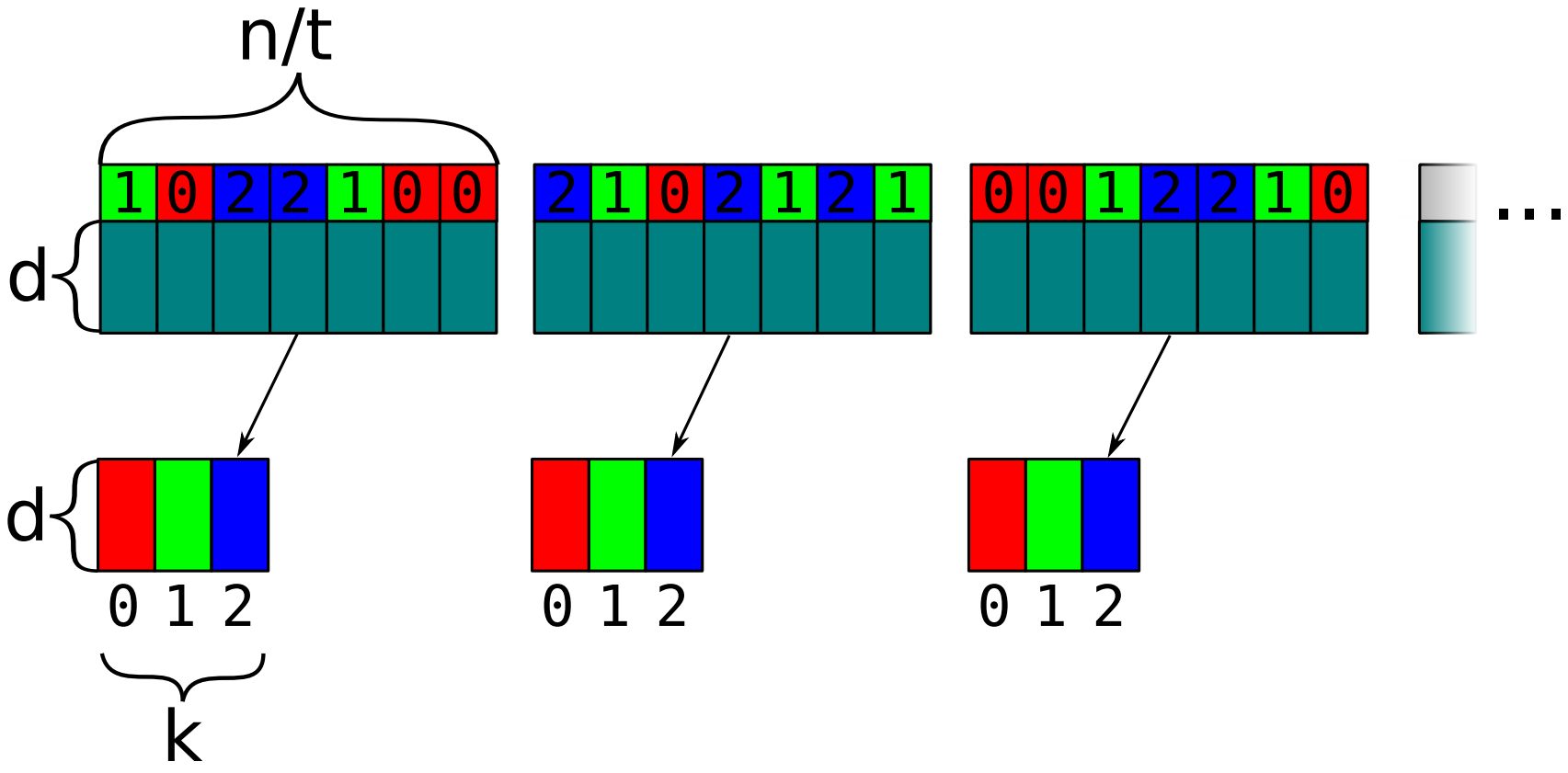
Computing Cluster Means: Just Right



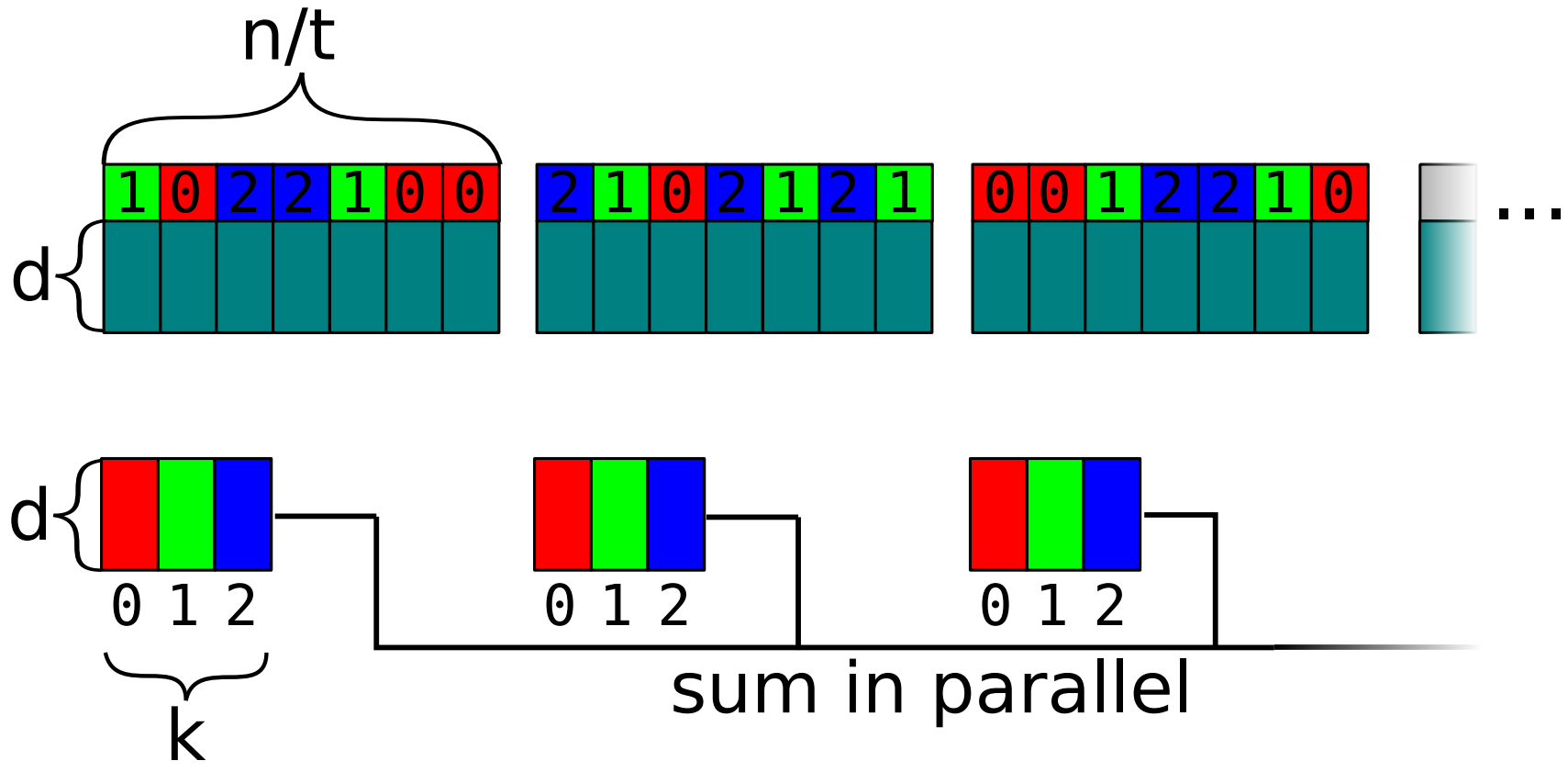
Computing Cluster Means: Just Right



Computing Cluster Means: Just Right



Computing Cluster Means: Just Right



Computing Cluster Sizes: Just Right

We use a Futhark language construct called a *stream reduction*.

```
let cluster_means_stream(cluster_sizes: [k]i32)
                        (points: [n][d]f32)
                        (membership: [n]i32): [k][d]f32 =
  stream_red
    matrix_add
    (λ(chunk_size:   i32)
      (points':      [chunk_size][d]f32)
      (membership': [chunk_size]i32) →
        cluster_means_seq cluster_sizes points' membership')
    points membership
```

For full parallelism, set chunk size to 1.

For full sequentialisation, set chunk size to n.

GPU Code Generation for `stream_red`

Broken up as:

```
let per_thread_results : [num_threads][k][d]f32 =
  ...
-- combine the per-thread results
let cluster_means =
  reduce (map (map (+)))
         (replicate k (replicate d 0.0))
         per_thread_results
```

The reduction with **map** (**map** (+)) is not great, as parallelism inside of a reduction operator cannot be exploited.
The compiler will recognise this structure and perform a transformation called *Interchange Reduce With Inner Map* (IRWIM); moving the reduction inwards at a cost of a transposition.

After IRWIM

We transform

```
let cluster_sizes =
  reduce (map (map (+)))
         (replicate k (replicate d 0.0))
         per_thread_results
```

and get

```
let per_thread_results' : [k][d][num_threads]f32 =
  rearrange (1,2,0) per_thread_results
let cluster_sizes =
  map (map (reduce (+) 0.0)) per_thread_results'
```

- **map** parallelism of size $k \times d$ - likely not enough.
- Futhark compiler generates a segmented reduction for **map** (**map** (**reduce** (+) 0.0)), which exploits also the innermost **reduce** parallelism.

Speedup on GPU of Chunked vs. Fully Parallel Implementation

- **Hardware**: NVIDIA Tesla K40
- **Input**: $k = 5; n = 10,000,000; d = 3.$
- **Fully parallel runtime:** 134.1ms
- **Chunked runtime:** 17.6ms

Speedup: $\times 7.6$

Improving Available Parallelism via Loop Distribution and Interchange

The Problem

Futhark permits *nested* (regular) parallelism, but GPUs prefer *flat* parallel kernels.

The Problem

Futhark permits *nested* (regular) parallelism, but GPUs prefer *flat* parallel kernels.

Solution: Have the compiler rewrite program to perfectly nested **map**s containing sequential code (or known parallel patterns such as segmented reduction), each of which can become a GPU kernel.

The Problem

Futhark permits *nested* (regular) parallelism, but GPUs prefer *flat* parallel kernels.

Solution: Have the compiler rewrite program to perfectly nested **map**s containing sequential code (or known parallel patterns such as segmented reduction), each of which can become a GPU kernel.

```
map (λxs → let y = reduce (+) 0 xs
           in map (+y) xs)
    xss
```

```
let ys = map (λxs → reduce (+) 0 xs) xss
in map (λxs y → map (+y) xs) xss ys
```

Previous Approaches

Full Flattening

- Always maximises parallelism, even when not worthwhile (e.g innermost loops in a matrix multiplication).
- Wasteful of memory (fully flattened matrix multiplication requires $O(n^3)$ space).
- Destroys access pattern information, rendering locality-of-reference optimisations hard or impossible.

Polyhedral approaches limited to affine programs and dependent on low-level analysis.

(See paper for more.)

Limited Flattening via Loop Fission

The classic map fusion rule:

$$\text{map } f \circ \text{map } g \Rightarrow \text{map } (f \circ g)$$

Limited Flattening via Loop Fission

The classic map fusion rule:

$$\text{map } f \circ \text{map } g \Rightarrow \text{map } (f \circ g)$$

We can also apply it backwards to obtain *fission*:

$$\text{map } (f \circ g) \Rightarrow \text{map } f \circ \text{map } g$$

This, along with other higher-order rules (see paper), are applied by the compiler to extract perfect map nests.

Example: (a) Initial program, we inspect the map-nest.

```
let (asss, bss) =
  map (λ(ps: [m]i32) →
         let ass = map (λ(p: i32): [m]i32 →
                          let cs = scan (+) 0 (iota p)
                          let r = reduce (+) 0 cs
                          in map (+r) ps) ps
         let bs = loop (ws=ps) for i < n do
                    map (λas w: i32 →
                           let d = reduce (+) 0 as
                           let e = d + w
                           in 2 * e) ass ws
         in (ass, bs)) pss
```

We assume the type of `pss : [m][m]i32`.

(b) Distribution.

```
let asss: [m][m][m]i32 =
  map (λ(ps: [m]i32) →
         let ass = map (λ(p: i32): [m]i32 →
                          let cs = scan (+) 0 (iota p)
                          let r = reduce (+) 0 cs
                          in map (+r) ps) ps
         in ass) pss
let bss: [m][m]i32 =
  map (λps ass →
         let bs = loop (ws=ps) for i < n do
                    map (λas w →
                           let d = reduce (+) 0 as
                           let e = d + w
                           in 2 * e) ass ws
         in bs) pss asss
```

(c) Interchanging outermost map inwards.

```
let asss: [m][m][m]i32 =
  map (λ(ps: [m]i32) →
         let ass = map (λ(p: i32): [m]i32 →
                          let cs = scan (+) 0 (iota p)
                          let r = reduce (+) 0 cs
                          in map (+r) ps) ps
         in ass) pss
let bss: [m][m]i32 =
  map (λps ass →
         let bs = loop (ws=ps) for i < n do
                    map (λas w →
                           let d = reduce (+) 0 as
                           let e = d + w
                           in 2 * e) ass ws
         in bs) pss asss
```

(c) Interchanging outermost map inwards.

```
let asss: [m][m][m]i32 =
  map (λ(ps: [m]i32) →
         let ass = map (λ(p: i32): [m]i32 →
                          let cs = scan (+) 0 (iota p)
                          let r = reduce (+) 0 cs
                          in map (+r) ps) ps
         in ass) pss
let bss: [m][m]i32 =
  loop (wss=pss) for i < n do
    map (λass ws →
           let ws' = map (λas w →
                            let d = reduce (+) 0 as
                            let e = d + w
                            in 2 * e) ass ws
           in ws') asss wss
```

(d) Skipping scalar computation.

```
let asss: [m][m][m]i32 =
  map (λ(ps: [m]i32) →
         let ass = map (λ(p: i32): [m]i32 →
                          let cs = scan (+) 0 (iota p)
                          let r = reduce (+) 0 cs
                          in map (+r) ps) ps
         in ass) pss
let bss: [m][m]i32 =
  loop (wss=pss) for i < n do
    map (λass ws →
           let ws' = map (λas w →
                            let d = reduce (+) 0 as
                            let e = d + w
                            in 2 * e) ass ws
           in ws') asss wss
```

(d) Skipping scalar computation.

```
let asss: [m][m][m]i32 =
  map (λ(ps: [m]i32) →
         let ass = map (λ(p: i32): [m]i32 →
                          let cs = scan (+) 0 (iota p)
                          let r = reduce (+) 0 cs
                          in map (+r) ps) ps
         in ass) pss
let bss: [m][m]i32 =
  loop (wss=pss) for i < n do
    map (λass ws →
           let ws' = map (λas w →
                            let d = reduce (+) 0 as
                            let e = d + w
                            in 2 * e) ass ws
           in ws') asss wss
```

(e) Distributing reduction..

```
let asss: [m][m][m]i32 =
  map (λ(ps: [m]i32) →
         let ass = map (λ(p: i32): [m]i32 →
                          let cs = scan (+) 0 (iota p)
                          let r = reduce (+) 0 cs
                          in map (+r) ps) ps
         in ass) pss
let bss: [m][m]i32 =
  loop (wss=pss) for i < n do
    map (λass ws →
           let ws' = map (λas w →
                            let d = reduce (+) 0 as
                            let e = d + w
                            in 2 * e) ass ws
           in ws') asss wss
```

(e) Distributing reduction.

```
let asss: [m][m][m]i32 =
  map (λ(ps: [m]i32) →
         let ass = map (λ(p: i32): [m]i32 →
                          let cs = scan (+) 0 (iota p)
                          let r = reduce (+) 0 cs
                          in map (+r) ps) ps
         in ass) pss
let bss: [m][m]i32 =
  loop (wss=pss) for i < n do
    let dss: [m][m]i32 =
      map (λass →
             map (λas →
                    reduce (+) 0 as) ass)
          asss
    in map (λws ds →
              let ws' =
                 map (λw d → let e  = d + w
                             in 2 * e) ws ds
              in ws') asss dss
```

(f) Distributing inner map.

```
let asss =
  map (λ(ps: [m]i32) →
         let ass = map (λ(p: i32): [m]i32 →
                          let cs = scan (+) 0 (iota p)
                          let r = reduce (+) 0 cs
                          in map (+r) ps) ps
         in ass) pss
let bss: [m][m]i32 = ...
```

(f) Distributing inner map.

```
let rss: [m][m]i32 =
  map (λ(ps: [m]i32) →
         let rss = map (λ(p: i32): i32 →
                          let cs = scan (+) 0 (iota p)
                          let r = reduce (+) 0 cs
                          in r) ps
         in rss) pss
let asss: [m][m][m]i32 =
  map (λ(ps: [m]i32) (rs: [m]i32) →
         map (λ(r: i32): [m]i32 →
                map (+r) ps) rs
      ) pss rss
let bss: [m][m]i32 = ...
```

(g) Cannot distribute as it would create irregular array.

```
let rss: [m][m]i32 =
  map (λ(ps: [m]i32) →
         let rss = map (λ(p: i32): i32 →
                          let cs = scan (+) 0 (iota p)
                          let r = reduce (+) 0 cs
                          in r) ps
         in rss) pss
let asss: [m][m][m]i32 = ...
let bss: [m][m]i32 = ...
```

Array `cs` has type `[p]i32`, and `p` is variant to the innermost map nest.

(h) These statements are sequentialised

```
let rss: [m][m]i32 =
  map (λ(ps: [m]i32) →
         let rss = map (λ(p: i32): i32 →
                          let cs = scan (+) 0 (iota p)
                          let r = reduce (+) 0 cs
                          in r) ps
         in rss) pss
let asss: [m][m][m]i32 = ...
let bss: [m][m]i32 = ...
```

Array `cs` has type `[p]i32`, and `p` is variant to the innermost map nest.

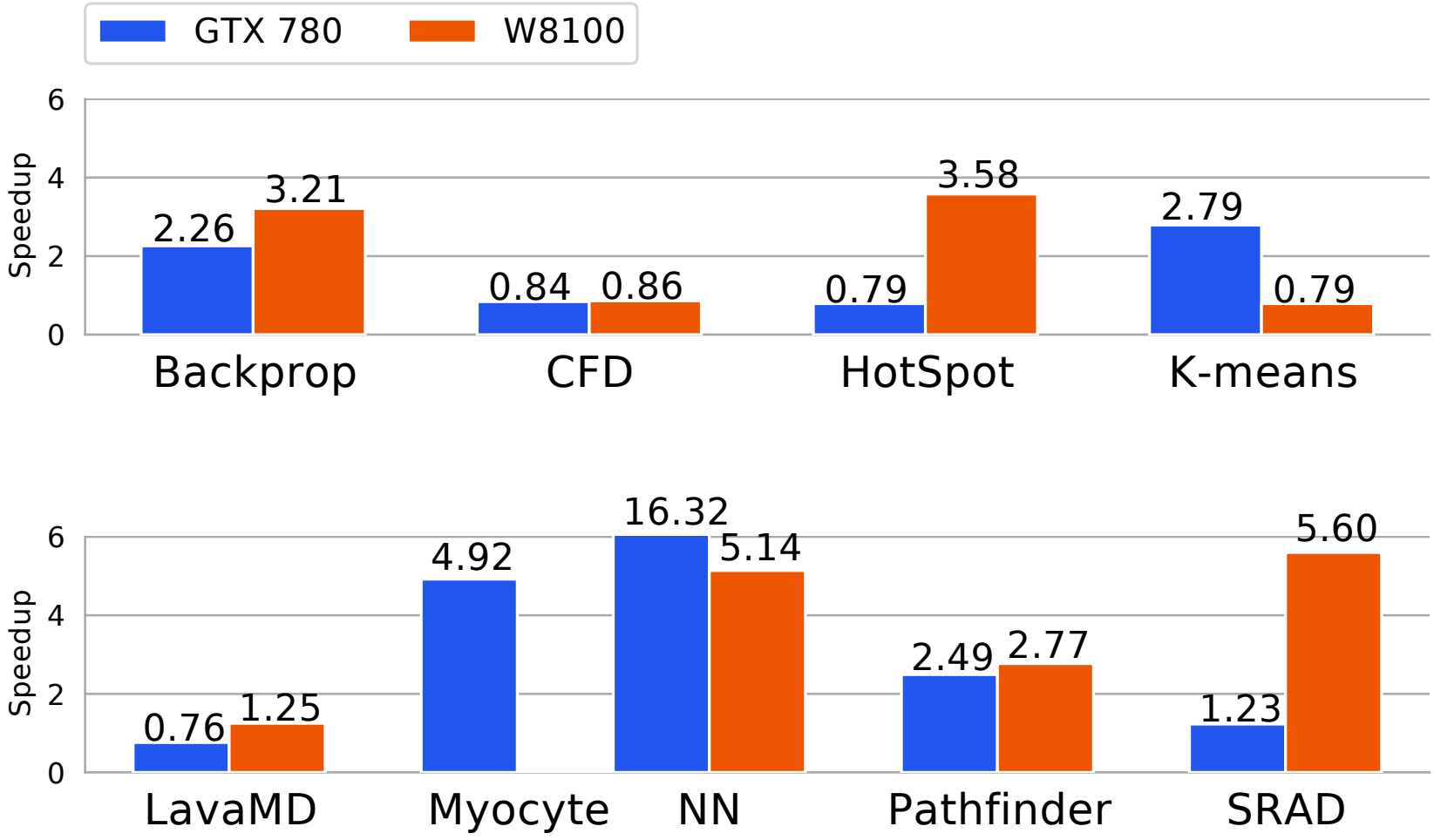
Result

```
let rss: [m][m]i32 = map (λps → map (...) ps) pss
let asss: [m][m][m]i32 =
  map (λps rs → map (λr → map (...) ps) rs) pss rss
let bss: [m][m]i32 =
  loop (wss=pss) for i < n do
    let dss: [m][m]i32 = map (λass → map (reduce ...) ass)
                             asss
    in map (λws ds → map (...) ws ds ) asss dss
```

From a single kernel with parallelism m to four kernels of parallelism m^2, m^3, m^3, and m^2.
The last two kernels are executed n times each.

Empirical Results

Speedup Over Hand-Written Rodinia OpenCL Code on NVIDIA GTX780 Ti and AMD W8100 GPUs



Conclusions

- Chunking data-parallel operators permit a balance between efficient sequential code with in-place updates and all necessary parallelism.
- Regular nested parallelism can be flattened via loop fission.
- Performance is decent.

Website `https://futhark-lang.org`

Code `https://github.com/HIPERFIT/futhark`

Benchmarks `https://github.com/HIPERFIT/futhark-pldi17`

Appendices

Validity of Chunking

Any fold with an associative operator $\odot$ can be rewritten as:

$$\texttt{fold} \odot \mathit{xs} = \texttt{fold} \odot (\texttt{map}\,(\texttt{fold} \odot)\,(\texttt{chunk}\,\mathit{xs}))$$

The trick is to provide a language construct where the user can provide a specialised implementation of the *inner* fold, which need not be parallel.

Optimising Locality of Reference

```
map (λx →
       let zs = map (+x) ys
       in reduce (+) 0 zs)
    xs
```

Only the outer **map** is parallel, and all threads are reading the same elements of ys, which can thus be cooperatively stored in GPU local memory by collective copying:

```
map (λx →
       let ys' = local ys
       let zs = map (+x) ys'
       in reduce (+) 0 zs)
    xs
```