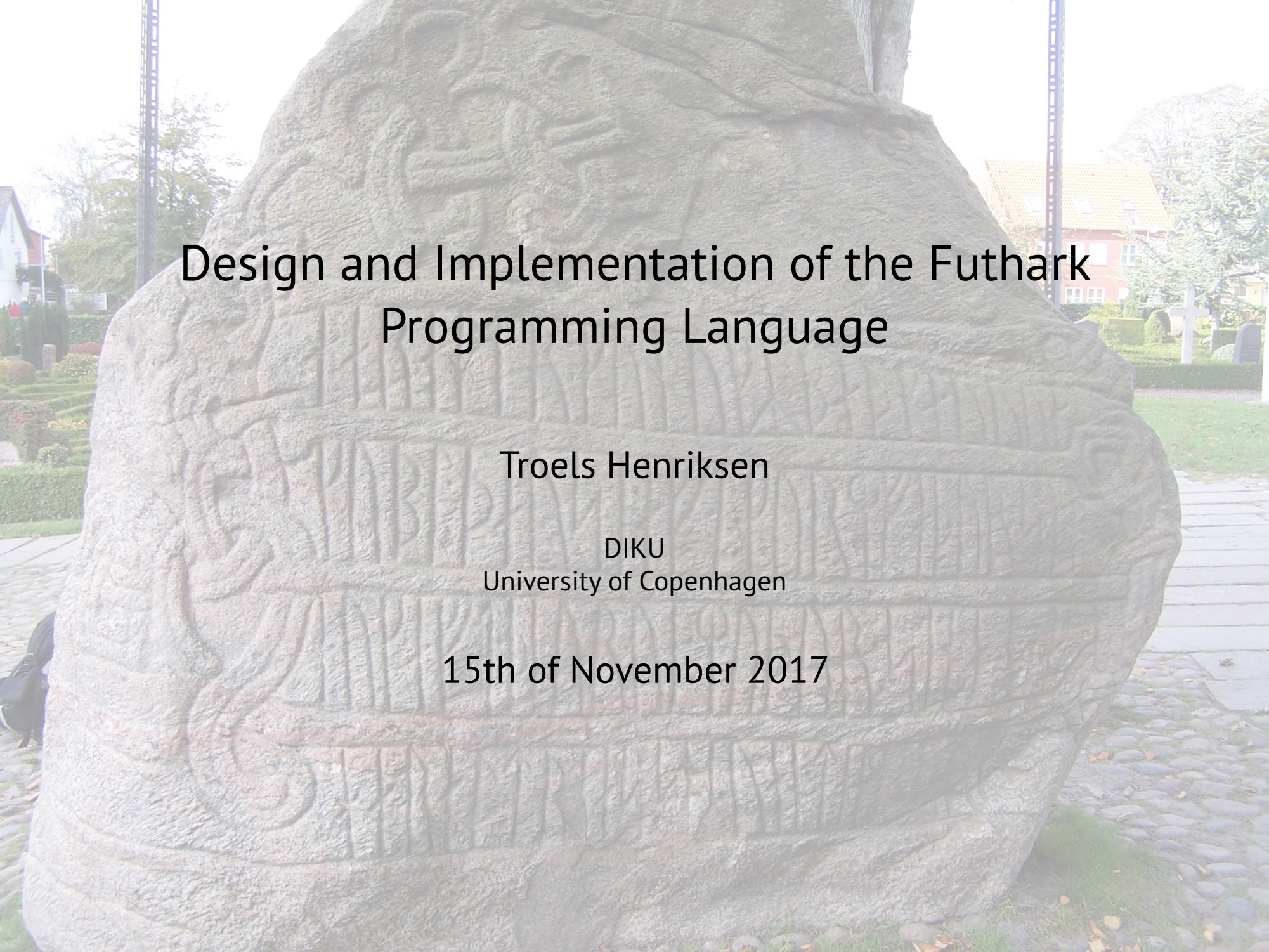


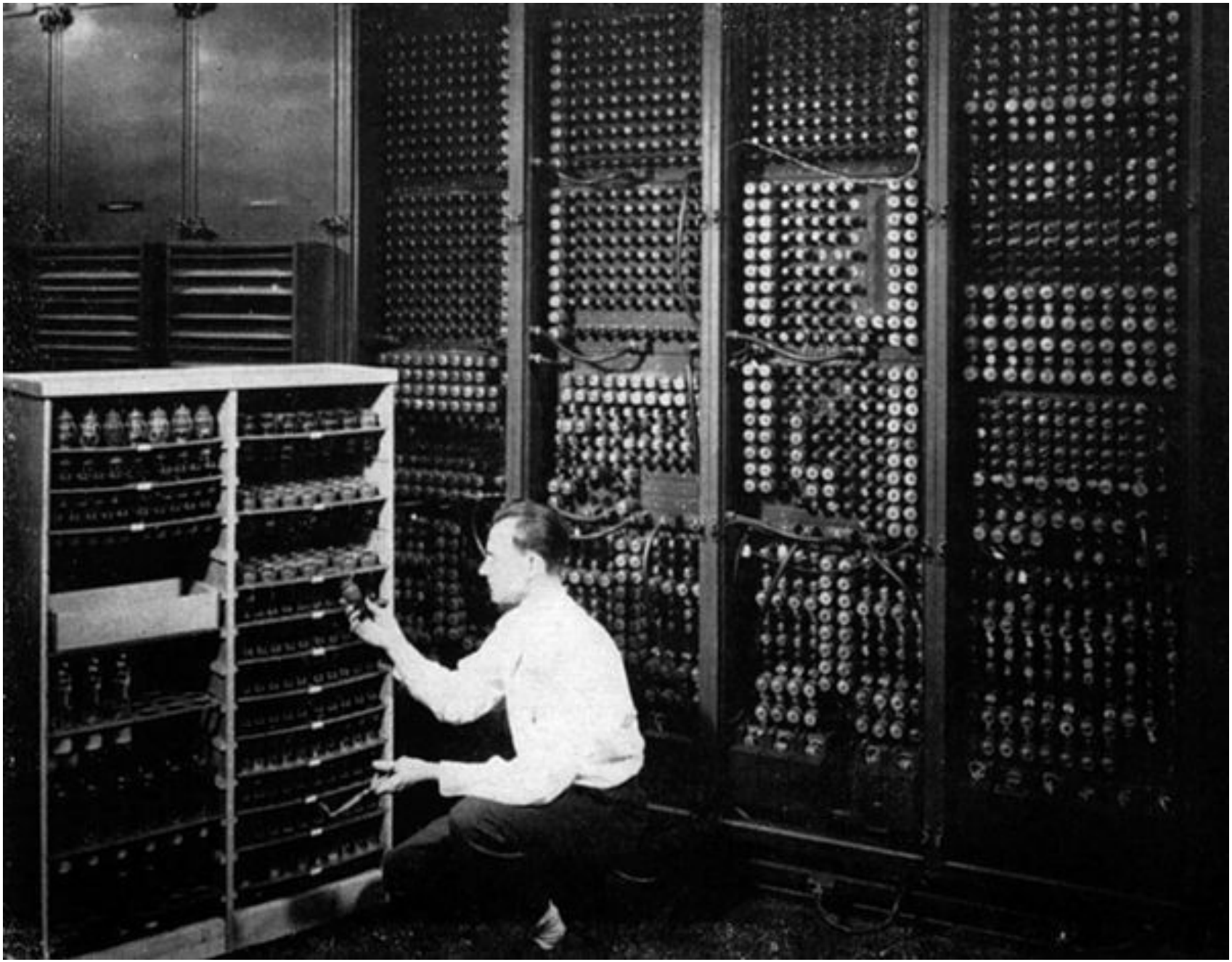
Design and Implementation of the Futhark Programming Language

Troels Henriksen

DIKU
University of Copenhagen

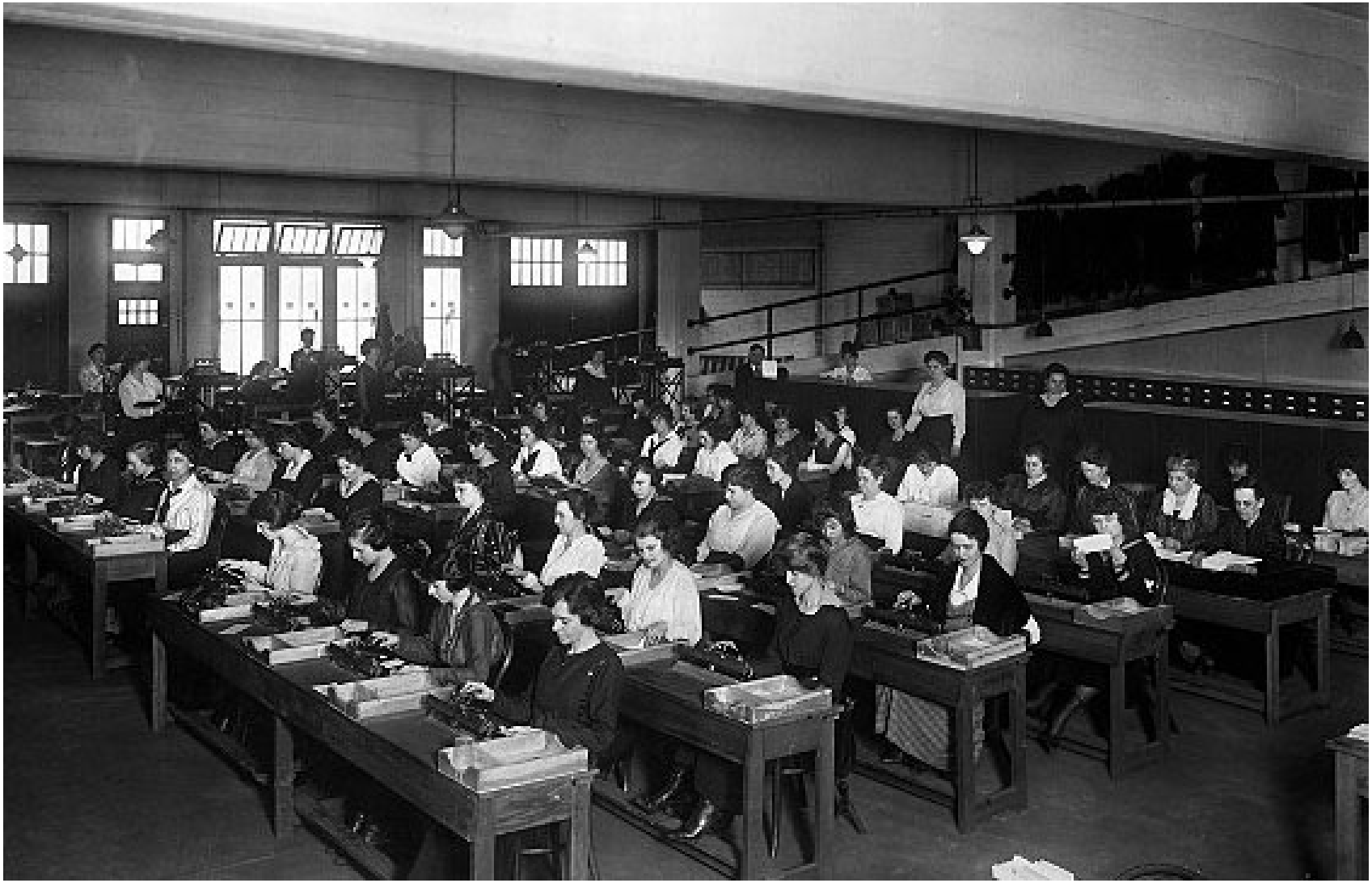
15th of November 2017

The first computers were not this

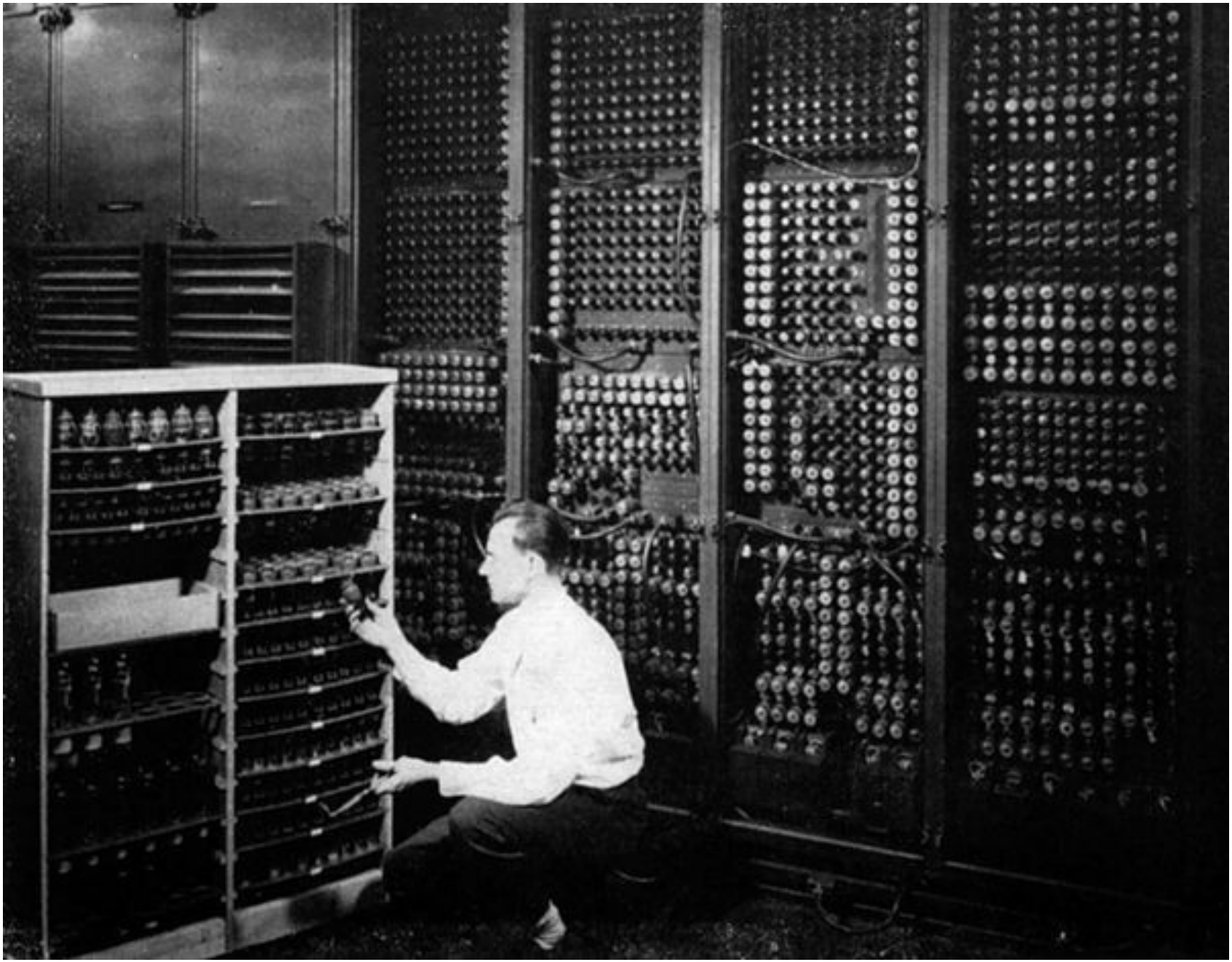


But this

And if you had a larger problem



But then they started looking like this



Then this

Then this

Then this

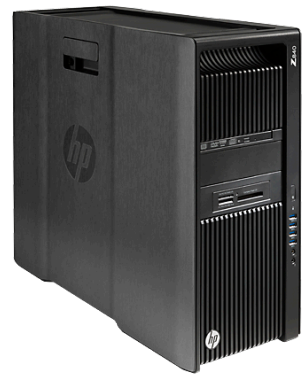
Then this

Then this

Then, from around 2005

Then, from around 2005

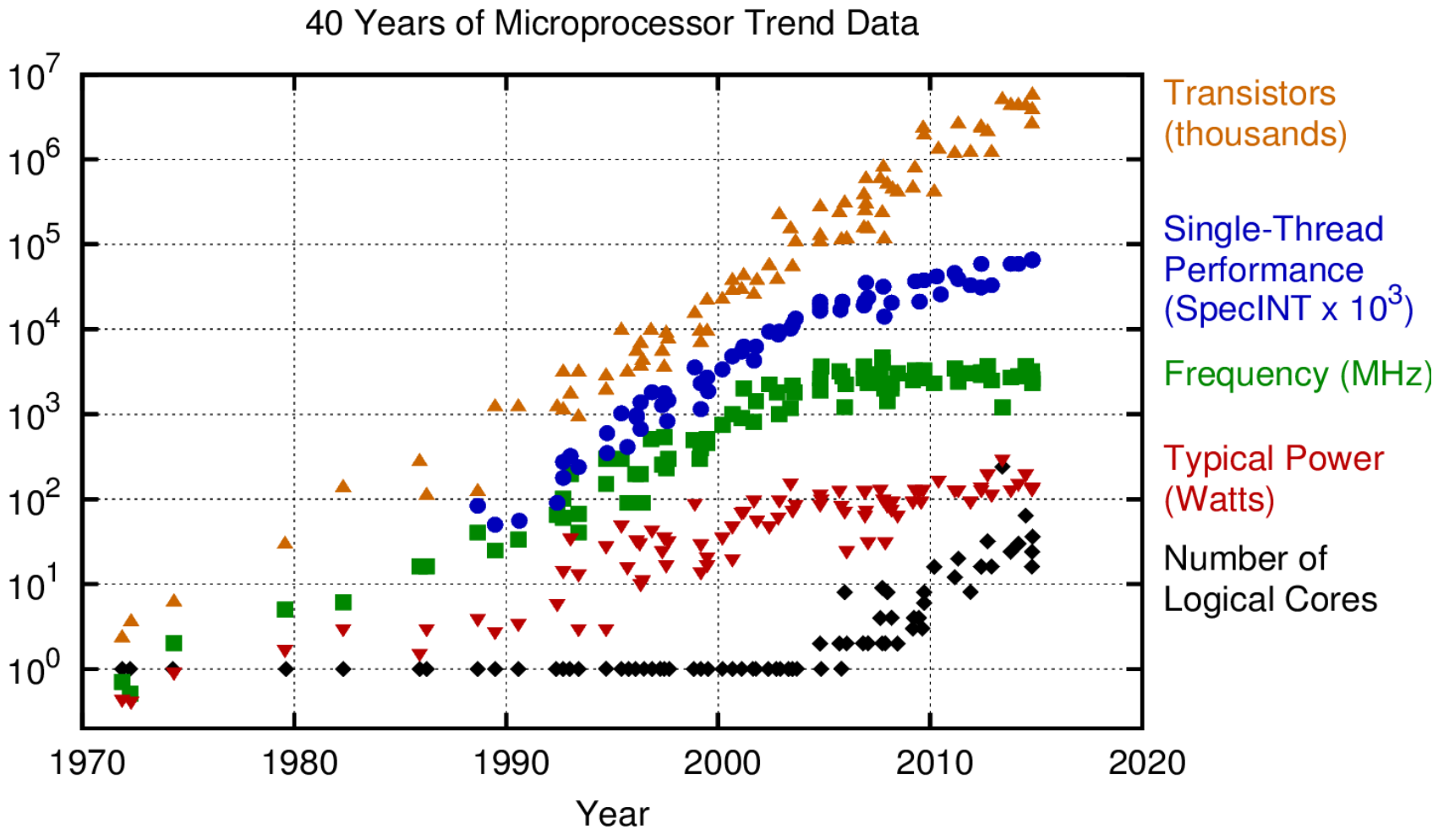




Then, from around 2005

Improvements in *sequential performance* stalled, although computers still got smaller and faster.

A graph of CPU progress

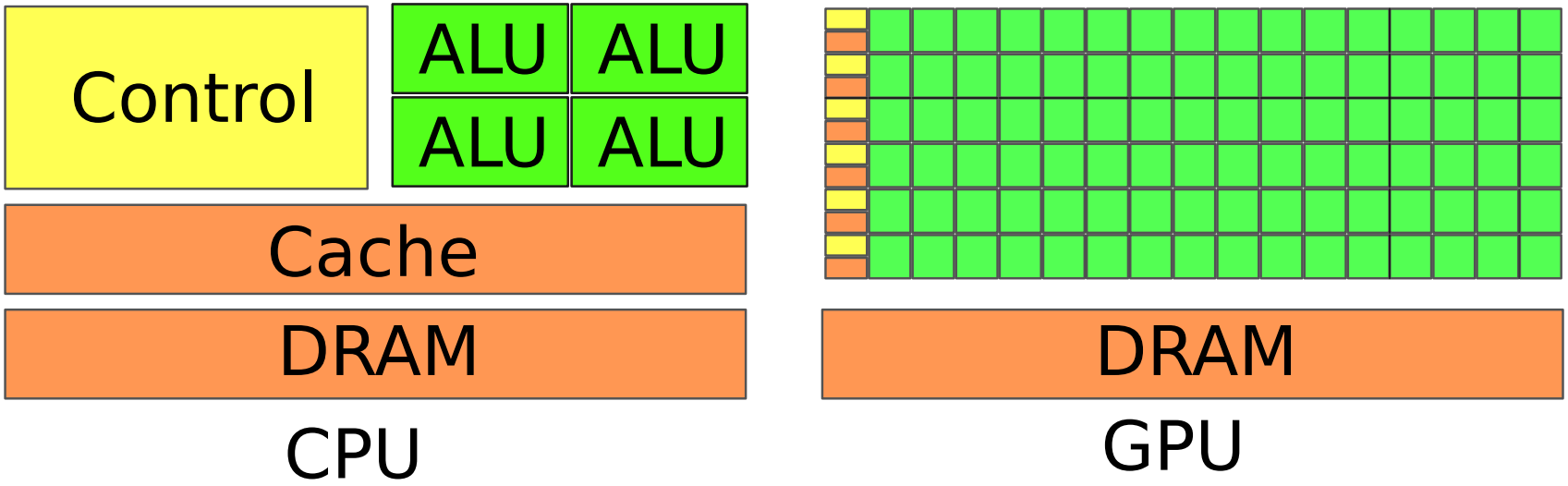


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

The Situation

- Transistors continue to shrink, so we can continue to build ever more advanced computers.
- CPU clock speed stalled around 3GHz in 2005, and improvements in sequential performance has been slow since then.
- Computers still get *faster*, but mostly for parallel code.
- General-purpose programming now often done on *massively parallel* processors, like Graphics Processing Units (GPUs).

GPUs vs CPUs



- GPUs have *thousands* of simple cores and taking full advantage of their compute power requires *tens of thousands* of threads.
- GPU threads are very *restricted* in what they can do: no stack, no allocation, limited control flow, etc.
- Potential *very high performance* and *lower power usage* compared to CPUs, but programming them is *hard*.

Massively parallel processing is currently a special case, but will be the common case in the future.

Two Guiding Quotes

When we had no computers, we had no programming problem either. When we had a few computers, we had a mild programming problem. Confronted with machines a million times as powerful, we are faced with a gigantic programming problem.

—Edsger W. Dijkstra (EWD963, 1986)

Two Guiding Quotes

When we had no computers, we had no programming problem either. When we had a few computers, we had a mild programming problem. Confronted with machines a million times as powerful, we are faced with a gigantic programming problem.

—Edsger W. Dijkstra (EWD963, 1986)

The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague.

—Edsger W. Dijkstra (EWD340, 1972)

Human brains simply cannot reason about concurrency on a massive scale

- We need a programming model with *sequential* semantics, but that can be *executed* in parallel.
- It must be *portable*, because hardware continues to change.
- It must support *modular* programming.

Sequential Programming for Parallel Machines

One approach: write imperative code like we've always done, and apply a *parallelising compiler* to try to figure out whether parallel execution is possible:

```
for (int i = 0; i < n; i++) {
  ys[i] = f(xs[i]);
}
```

Is this parallel? **Yes.** But it requires careful inspection of read/write indices.

Sequential Programming for Parallel Machines

What about this one?

```
for (int i = 0; i < n; i++) {
  ys[i+1] = f(ys[i], xs[i]);
}
```

Yes, but hard for a compiler to detect.

- Many algorithms are innately parallel, but phrased sequentially when we encode them in current languages.
- A *parallelising compiler* tries to reverse engineer the original parallelism from a sequential formulation.
- Possible in theory, is called *heroic effort* for a reason.

Why not use a language where we can just say exactly what we mean?

Functional Programming for Parallel Machines

Common purely functional combinators have *sequential semantics*, but permit *parallel execution*.

```
for (int i = 0;          ~  let ys = map f xs
     i < n;
     i++) {
  ys[i] = f(xs[i]);
}
```

```
for (int i = 0;          ~  let ys = scan f xs
     i < n;
     i++) {
  ys[i+1] = f(ys[i], xs[i]);
}
```

Why This Was Worth a PhD Project

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

Why This Was Worth a PhD Project

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 is a high-level language!

- Futhark is *not* a "GPU language"—it is a hardware-agnostic parallel language.
- However, we have written a Futhark *compiler* that can generate good GPU code.
- The compiler is *not* a "parallelising compiler". The parallelism is *explicitly given* by the programmer. The compiler's job is to figure out what to do with it. It's more of a *sequentialising compiler*.
- *Co-design* between compiler and language, inspired by hand-written code for target hardware.
- GPUs are a challenging target, so if Futhark can be translated to good GPU code, then it can probably also be translated to good CPU code.

I will talk about key language properties and compilation techniques for generating good GPU code.

Futhark at a Glance

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

- **Data-parallel loops**

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

- **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)
```

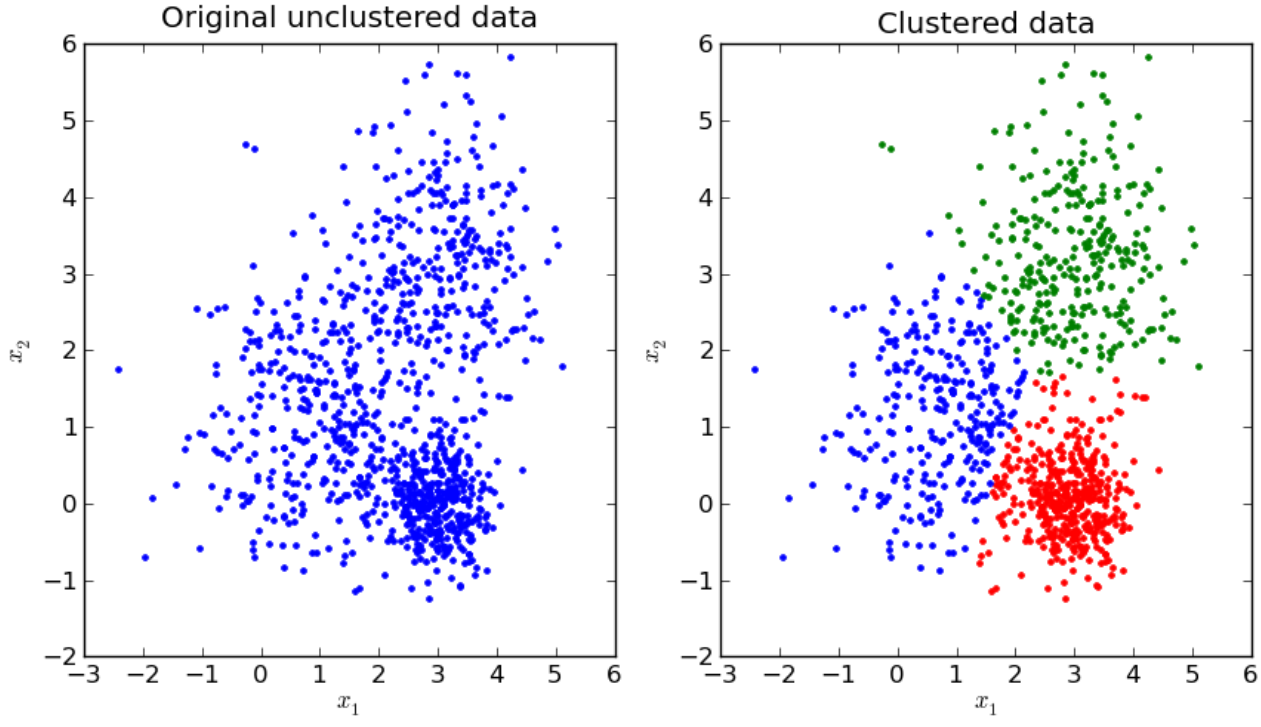
CASE STUDY: *k*-MEANS CLUSTERING

Let's do it.

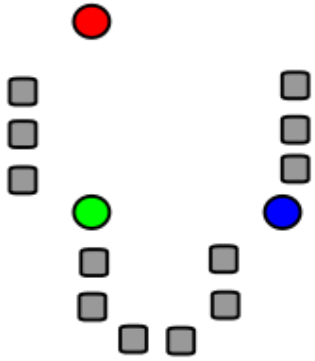
—Sonic the Hedgehog (Sonic Rush, 2005)

The Problem

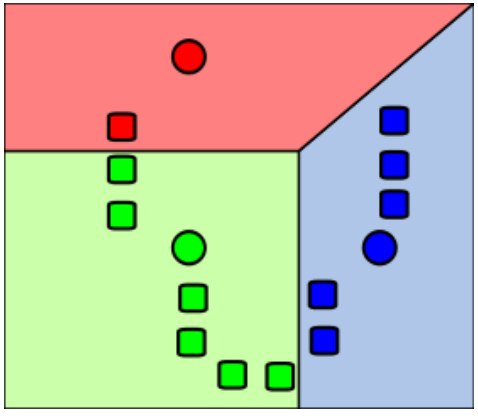
We are given n points in some d-dimensional space, which we must partition into k disjoint sets, such that we minimise the inter-cluster sum of squares (the distance from every point in a cluster to the centre of the cluster).
Example with $d = 2, k = 3, n =$ *more than I can count*:



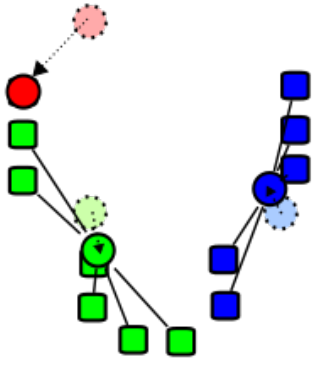
The Solution (from Wikipedia)



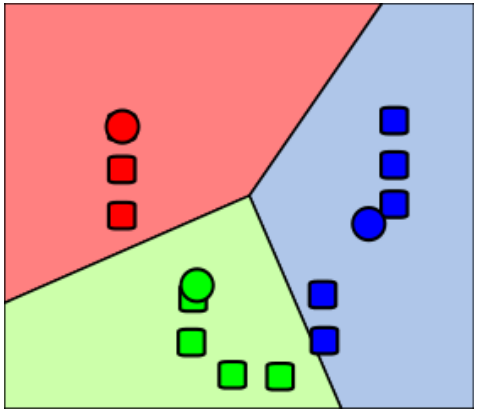
(1) k initial "means" (here $k = 3$) are randomly generated within the data domain.



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

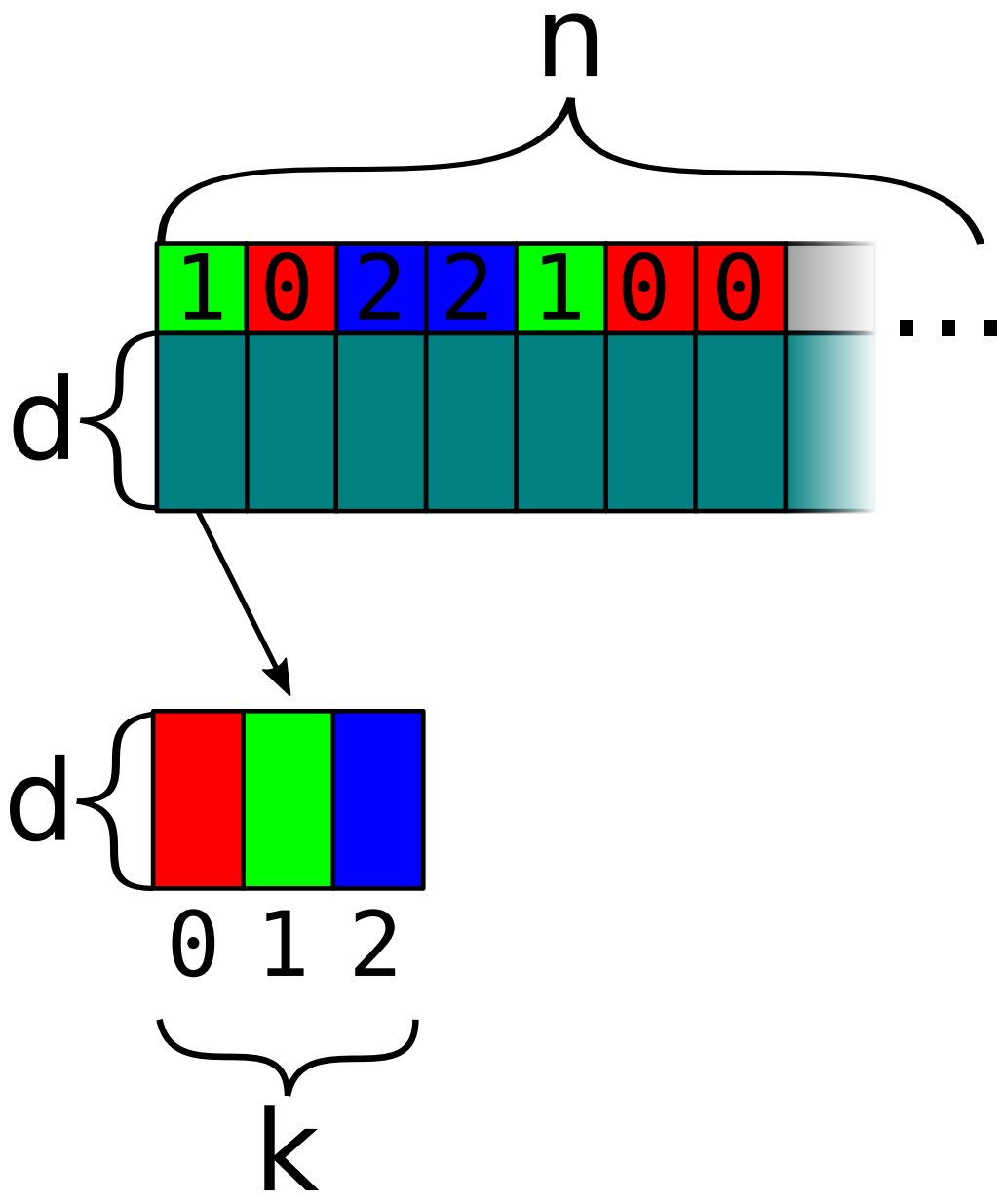


(3) The centroid of each of the k clusters becomes the new mean.

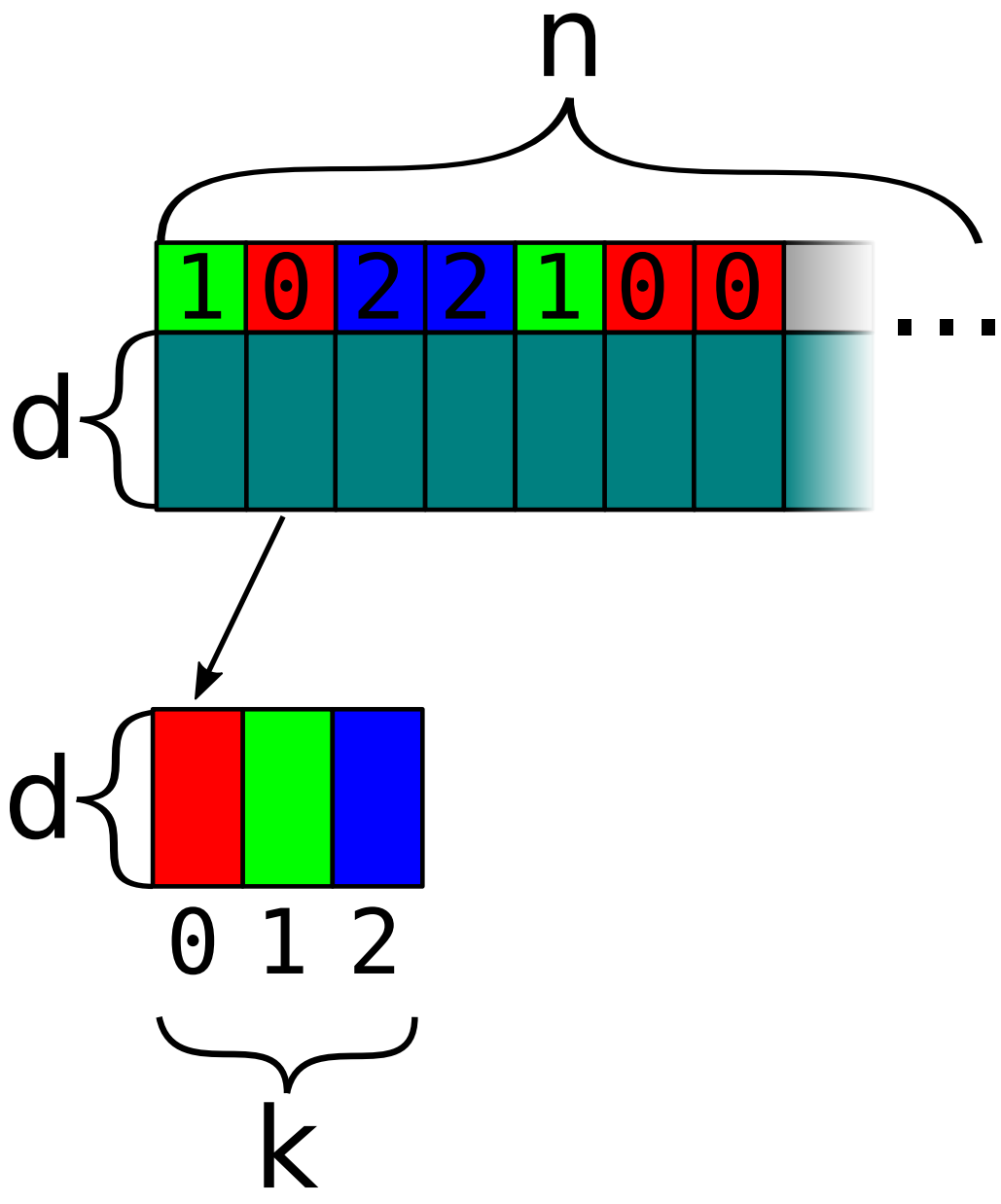


(4) Steps (2) and (3) are repeated until convergence has been reached.

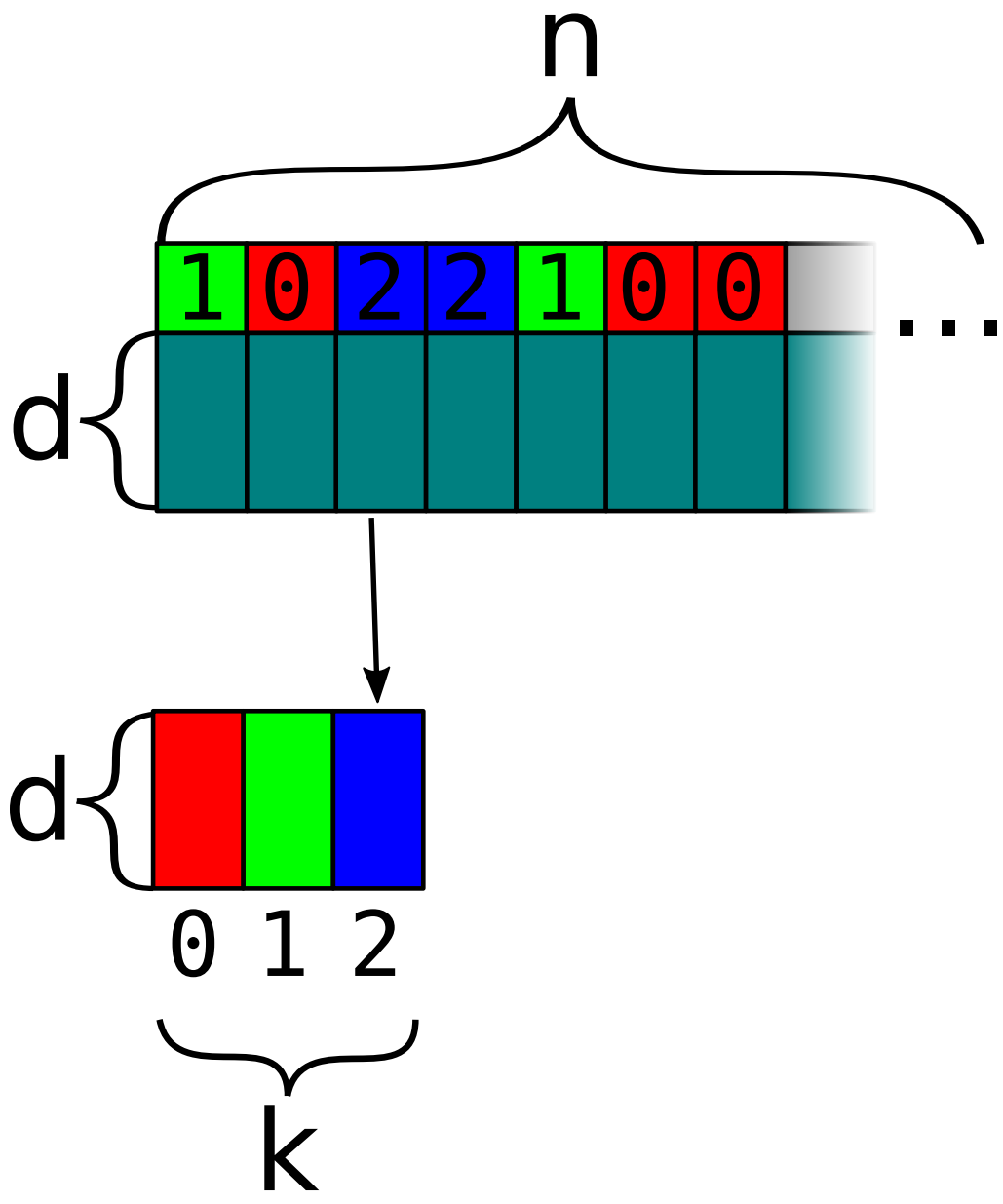
Computing Cluster Means: Fully Sequential



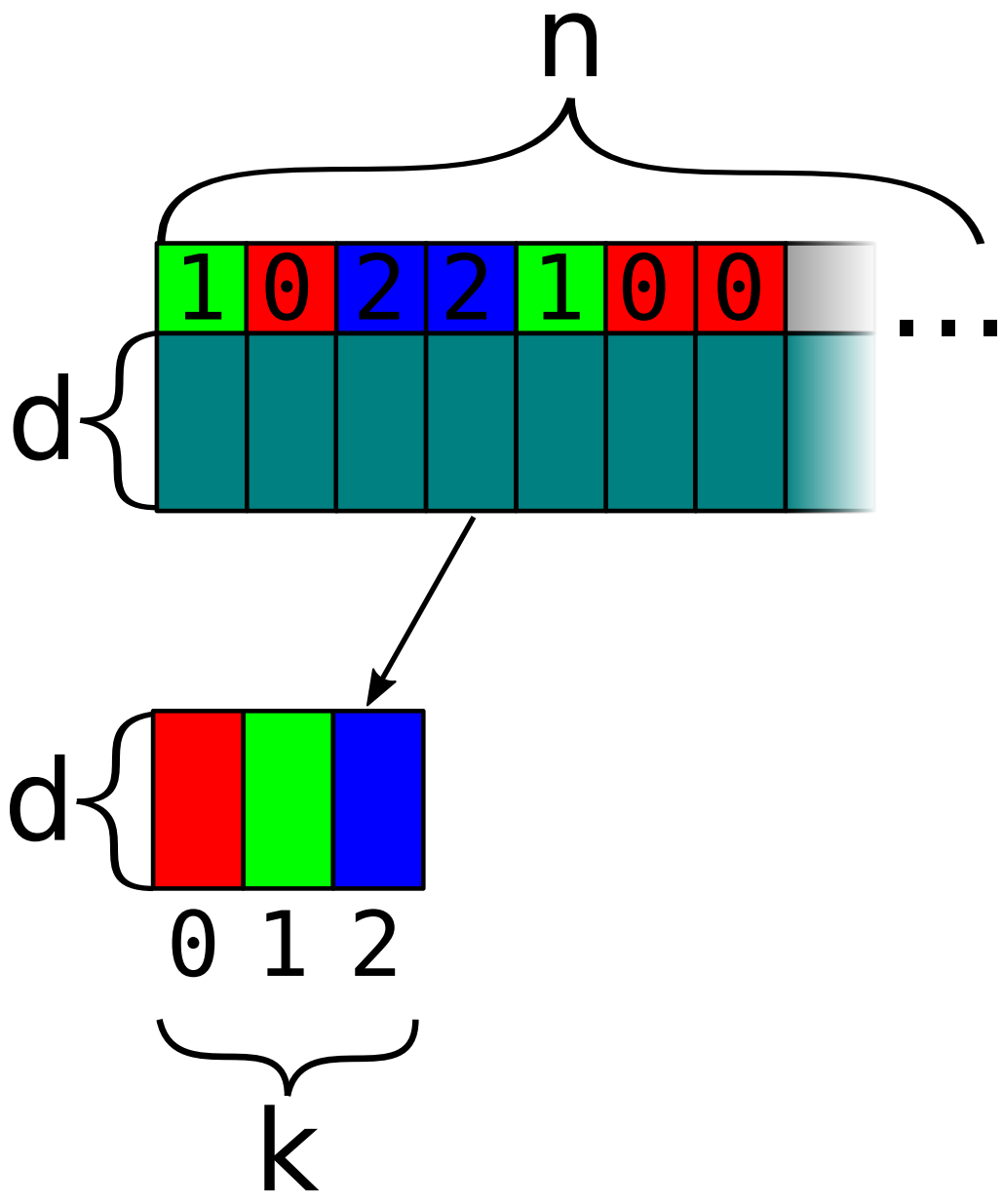
Computing Cluster Means: Fully Sequential



Computing Cluster Means: Fully Sequential



Computing Cluster Means: Fully Sequential



Computing Cluster Means: Fully Sequential

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

let cluster_means_seq [k][n][d]
                      (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'
```

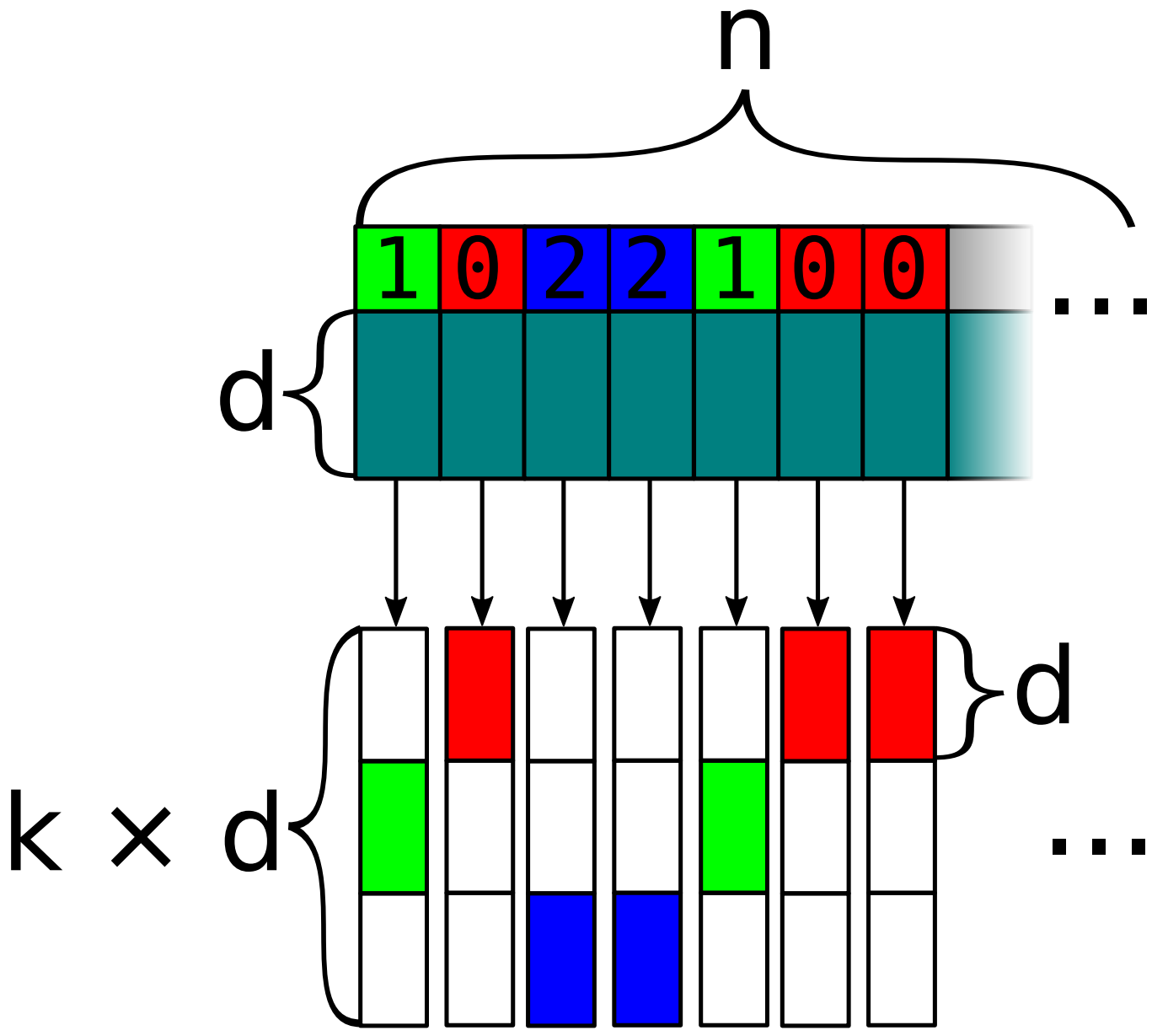
Computing Cluster Means: Fully Sequential

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

let cluster_means_seq [k][n][d]
                      (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'
```

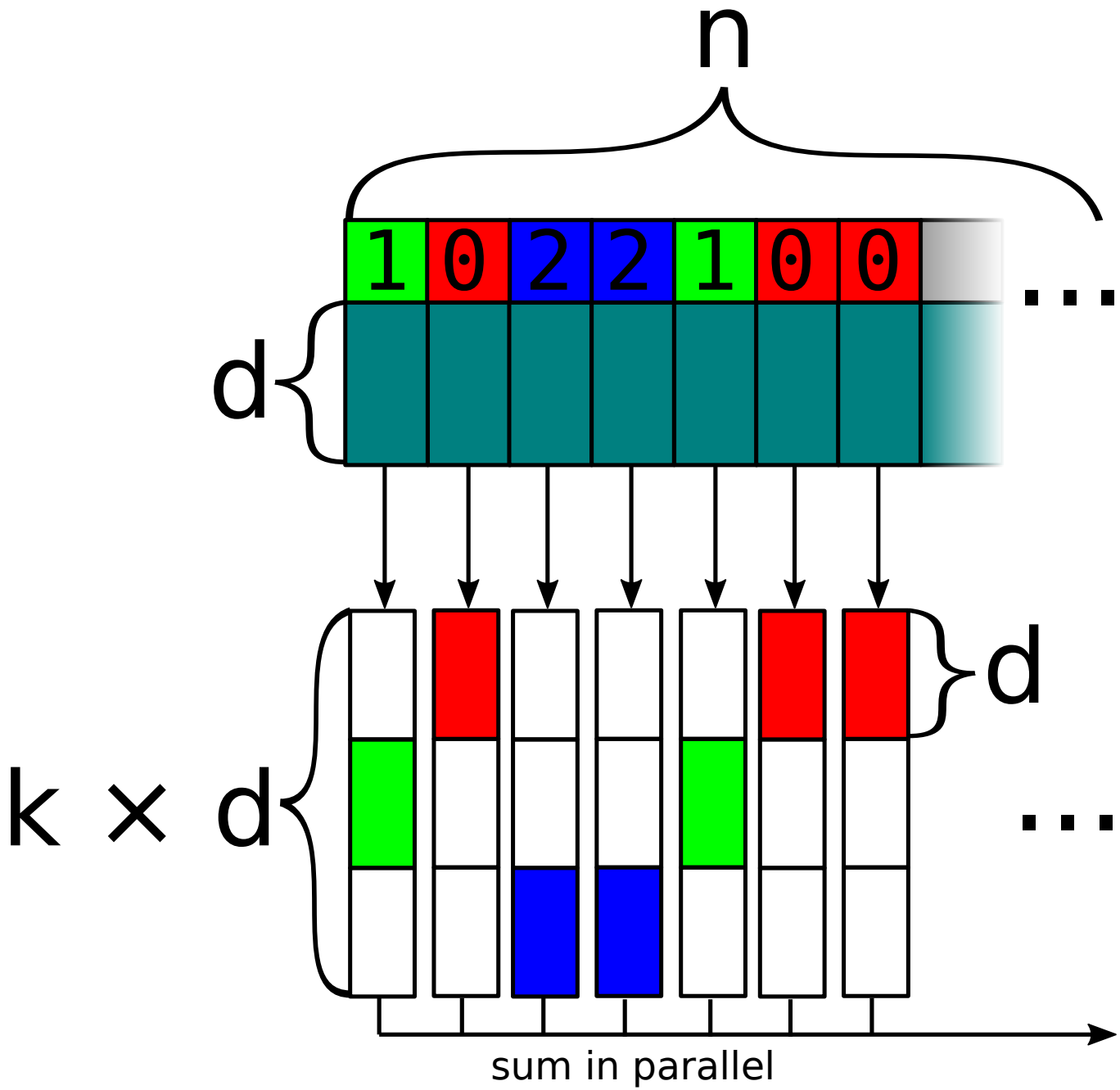
Problem

$O(n \times d)$ 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 [k][d]
               (xss: [k][d]f32) (yss: [k][d]f32): [k][d]f32 =
  map (λxs ys → map (+) xs ys) xss yss

let cluster_means_par [k][n][d]
                      (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.

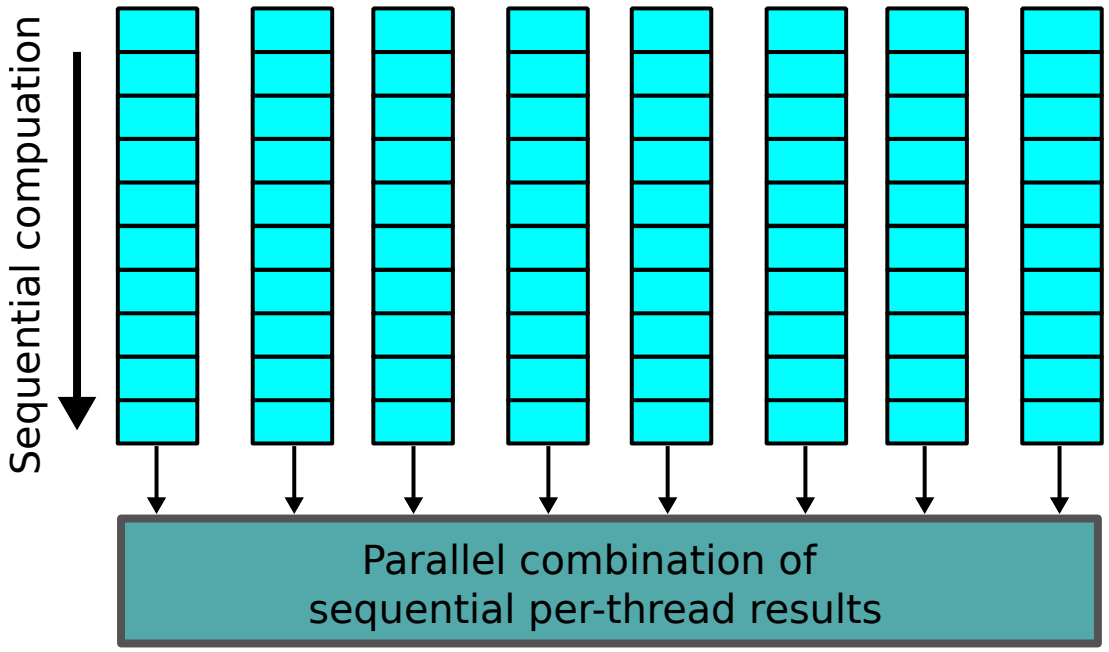
```
let matrix_add [k][d]
               (xss: [k][d]f32) (yss: [k][d]f32): [k][d]f32 =
  map (λxs ys → map (+) xs ys) xss yss

let cluster_means_par [k][n][d]
                      (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.

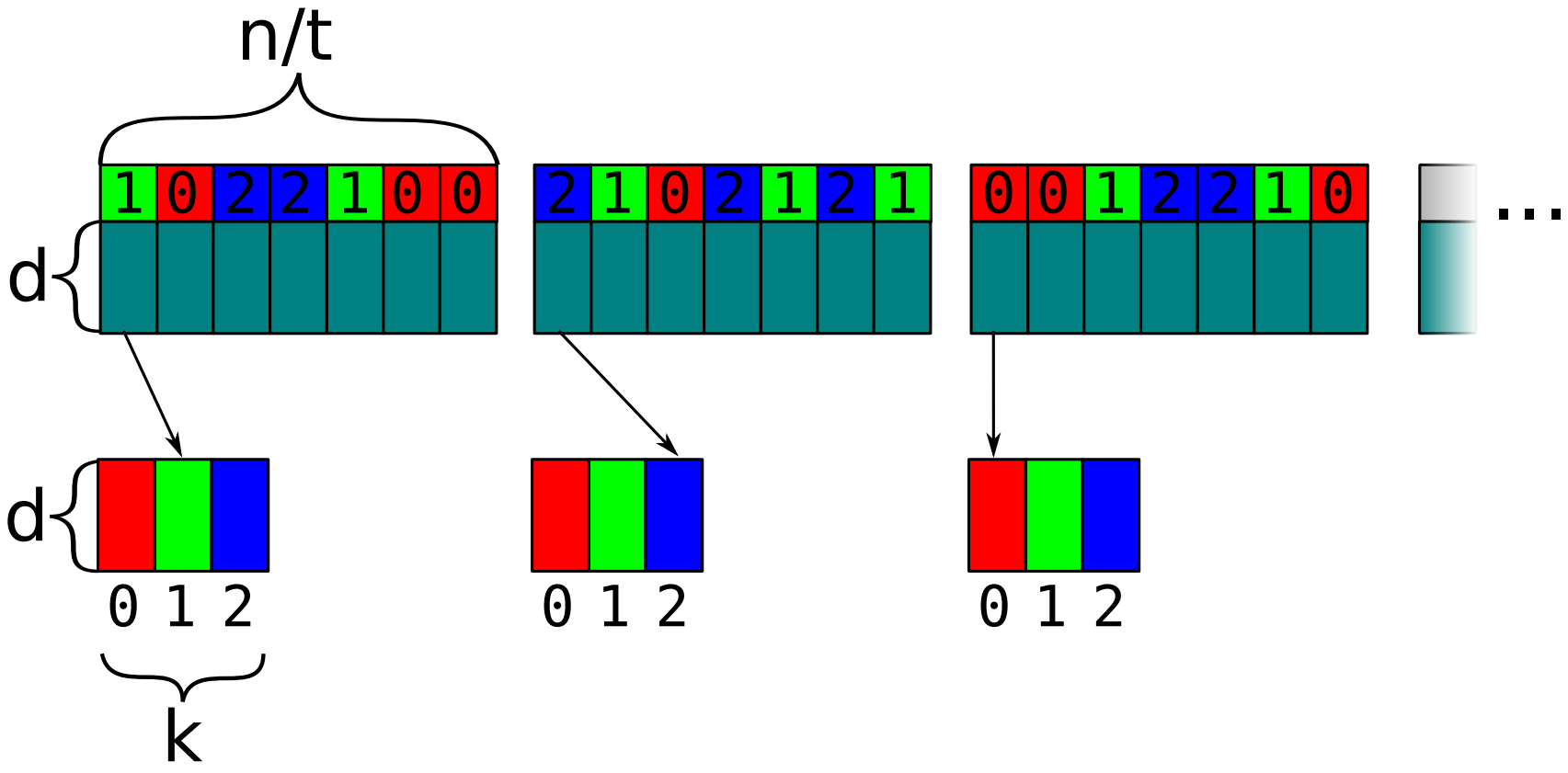
One Futhark Design Principle

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.

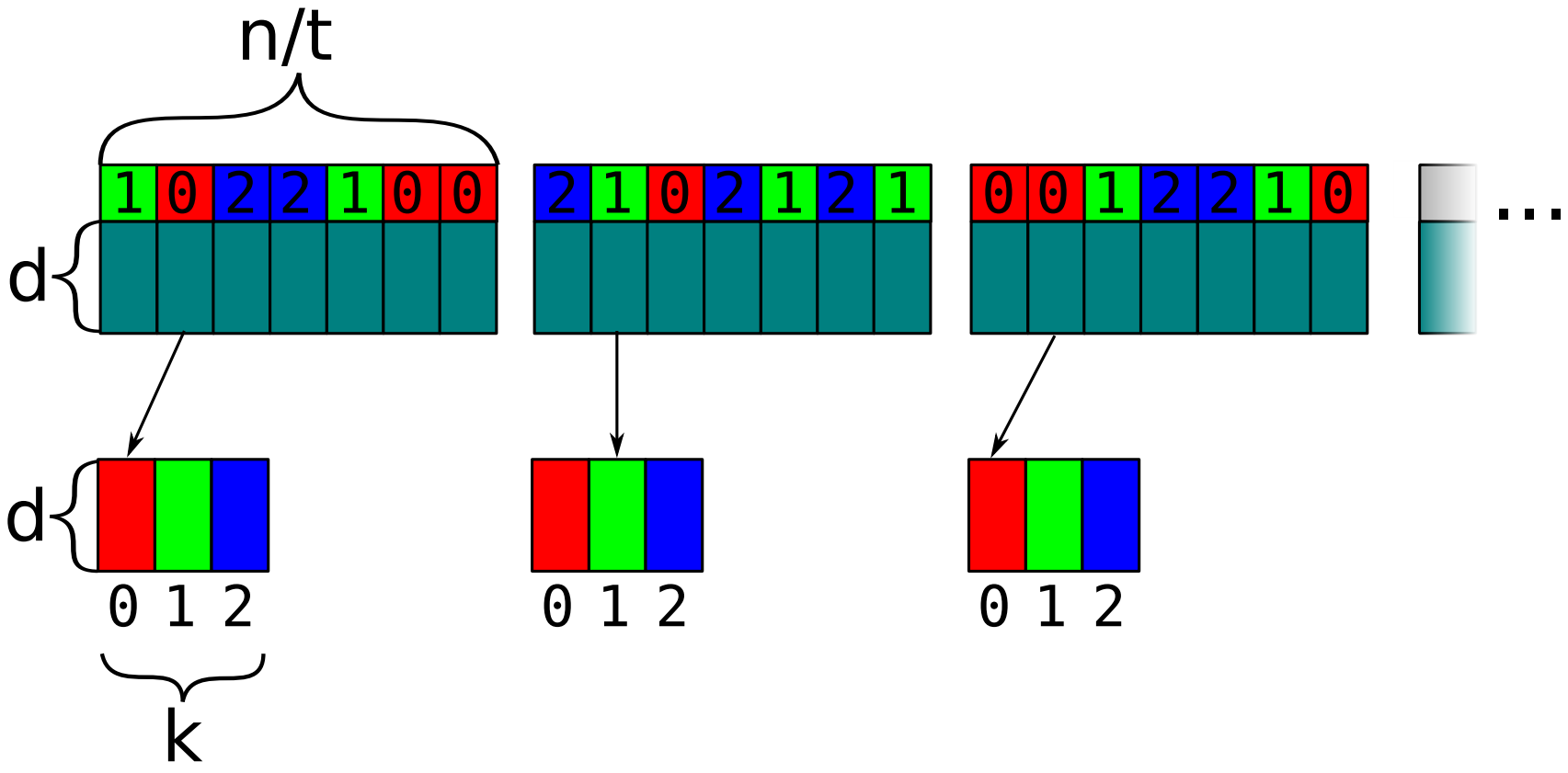


The optimal number of threads varies from case to case, so this should be abstracted from the programmer.

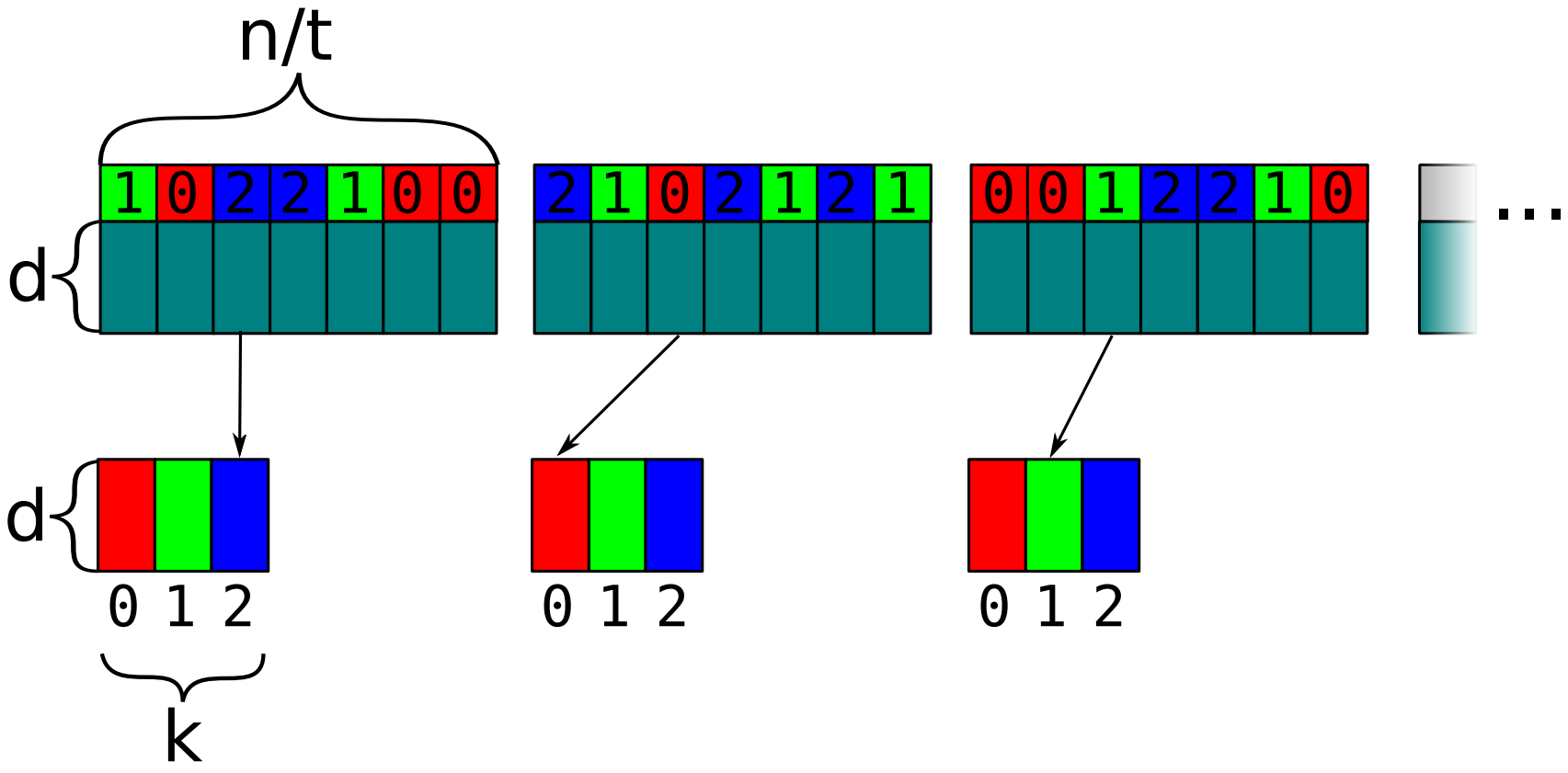
Computing Cluster Means: Just Right



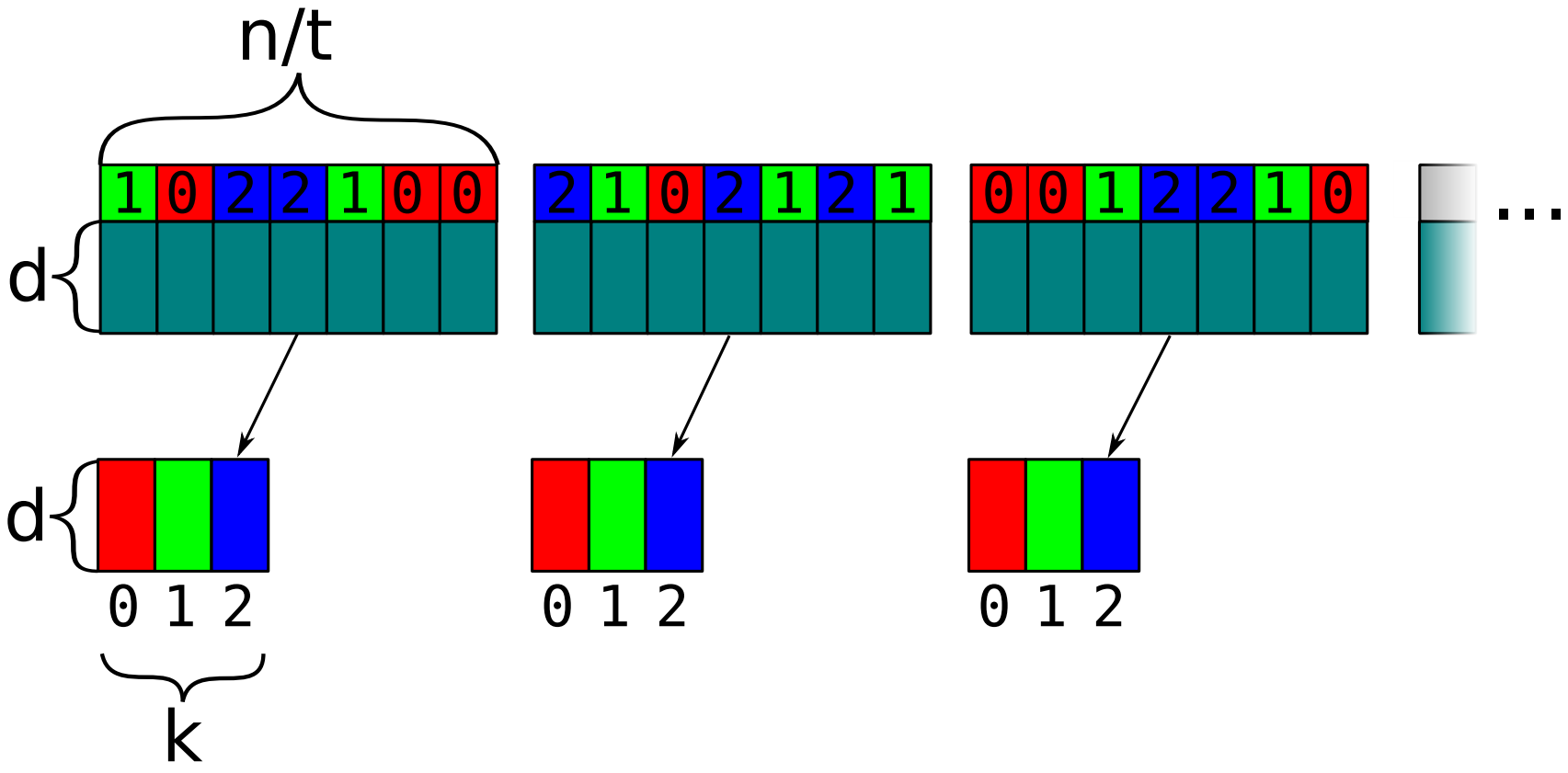
Computing Cluster Means: Just Right



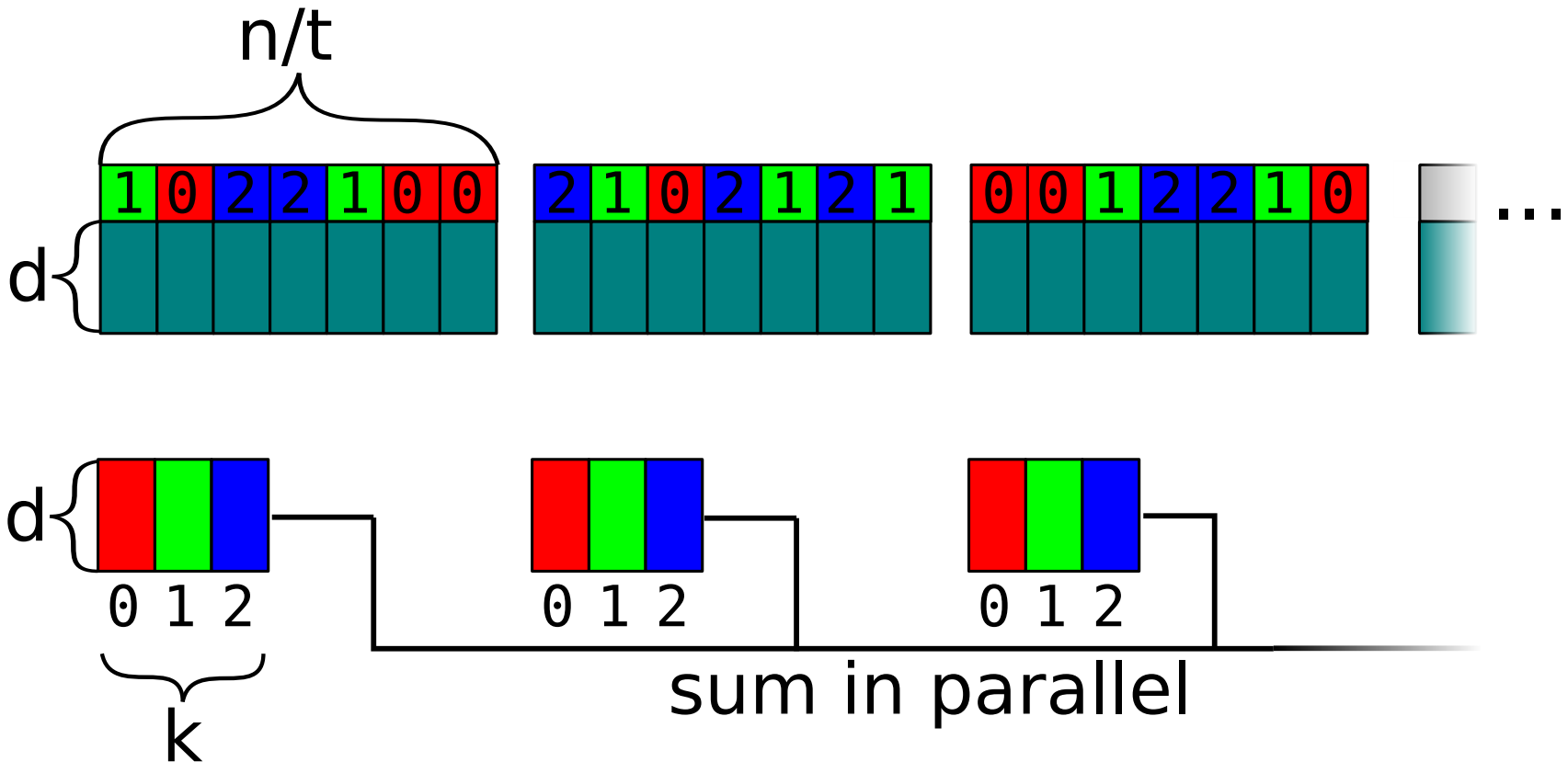
Computing Cluster Means: Just Right



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 [k][n][d]
                         (cluster_sizes: [k]i32)
                         (points: [n][d]f32)
                         (membership: [n]i32): [k][d]f32 =
  stream_red
    matrix_add
    (λ[chunk_size]
      (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_means =
  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.
- The 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

Oh, look! It changed shape! Did you see that?!
—Miles "Tails" Prower (Sonic Adventure, 1998)

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
```

The Universal Approach: Full Flattening

- Developed by Guy Blelloch in the early 90s.
- Able to exploit *all* parallelism.
- 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.

We want to exploit the parallelism that is *profitable*, but not 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 thesis), 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 VALIDATION

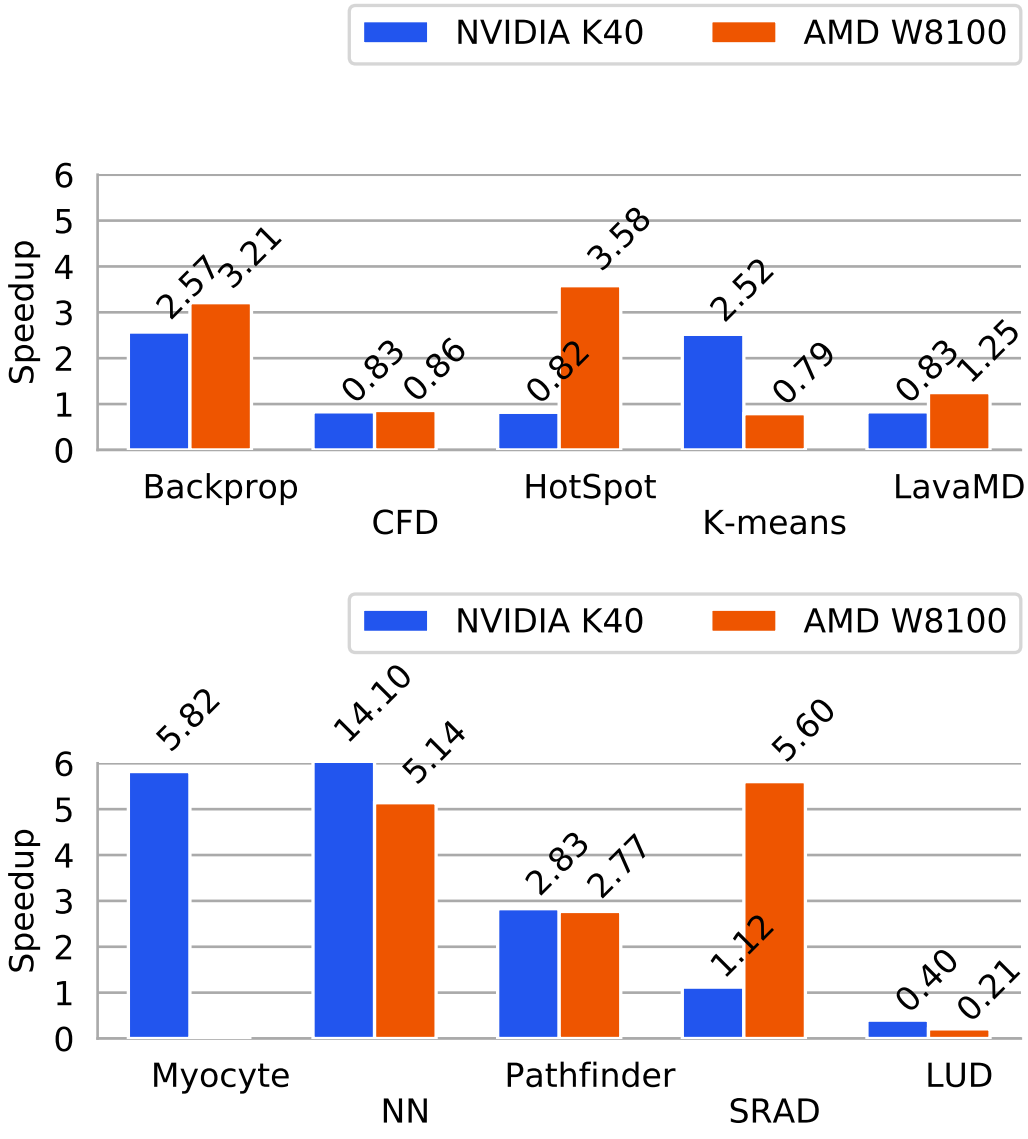
I'll show you what true speed REALLY is!
—Sonic the Hedgehog (Sonic Riders, 2006)

Empirical validation

The Question: Is it possible to construct a purely functional hardware-agnostic programming language that is convenient to use and provides good parallel performance?
Hard to Prove: Only performance is easy to quantify, and even then…

- No good objective criterion for whether a language is “fast”.
- Best practice is to take benchmark programs written in other languages, port or re-implement them, and see how they behave.
- Most benchmarks (16) originally written in low-level CUDA or OpenCL, but a few (5) are from Accelerate, another high-level parallel language.

Rodinia



- CUDA and OpenCL implementations of widely varying quality.
- This makes them “realistic”, in a sense.

Rodinia: HotSpot — a stencil application
0.82× "speedup"

At a high level, the stencil is a sequential loop containing a nested **map** over the $m \times m$ iteration space:

```
loop s = s0 for i < n do
  let s' = map (λi → map (f s i) [0...m−1]) [0...m−1]
  in s'
```

- Futhark performs a copy of the intermediate `s'` array for every iteration, while the reference implementation does pointer swapping.
- These copies account for 30% of runtime; which is why the reference implementation is slightly faster.

Rodinia: SRAD — a stencil application with some differential equations

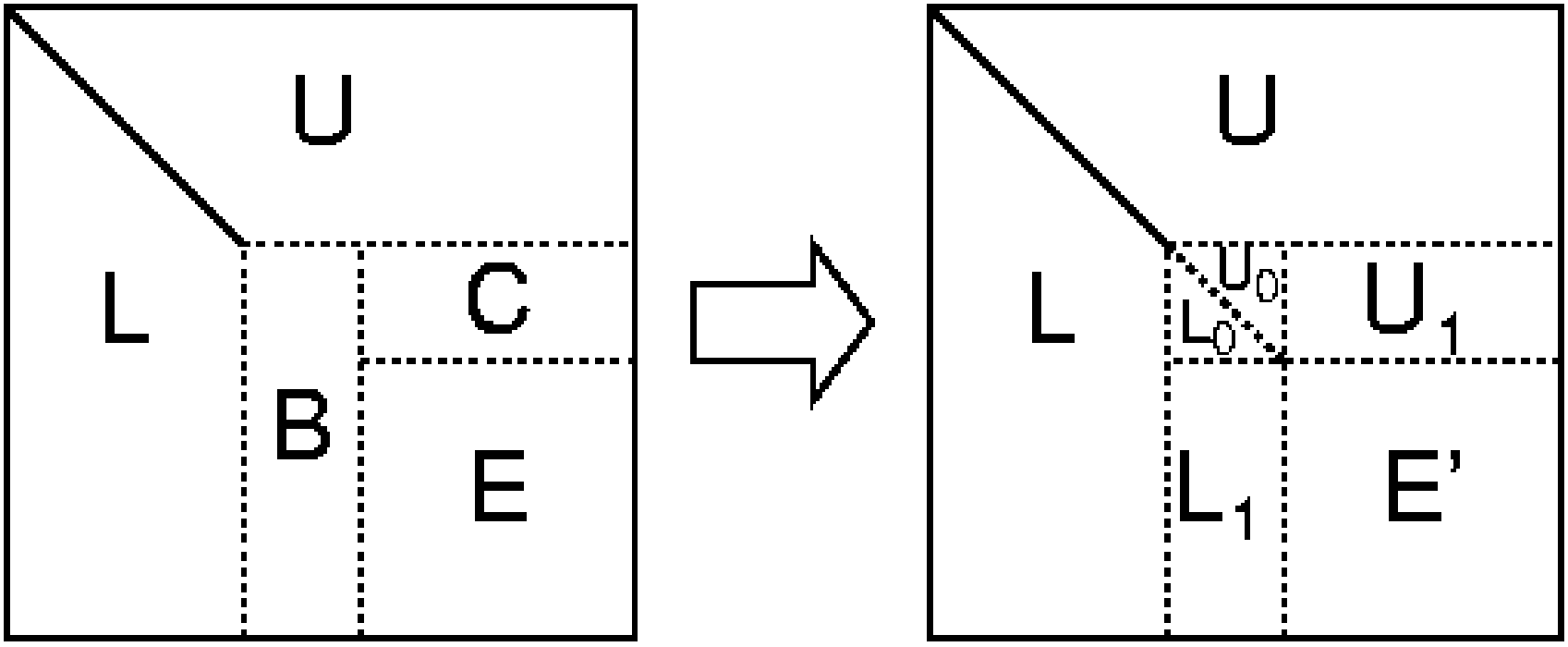
1.12× speedup

Same rough pattern as HotSpot:

```
loop s = s0 for i < n do
  let x = reduce (+) 0 (reshape (m*m) s)
  let s' = map (λi → map (f s i) [0...m-1]) [0...m-1]
  in s'
```

- But note the reduction!
- They are tedious to implement efficiently, and the reference implementation contains a naive implementation that loses maybe 30% potential performance.
- The Futhark compiler does not get bored, and performs even the tedious optimisations, which is enough to win here.
- Humans are good at the big picture, and compilers can resolve the details — and they do add up.

Rodinia: LUD



(Illustration from *Software Libraries for Linear Algebra Computations on High Performance Computers*, by Jack J. Dongarra and David W. Walker.)

Rodinia: LUD — computing the block
0.40× speedup

A loop that is roughly

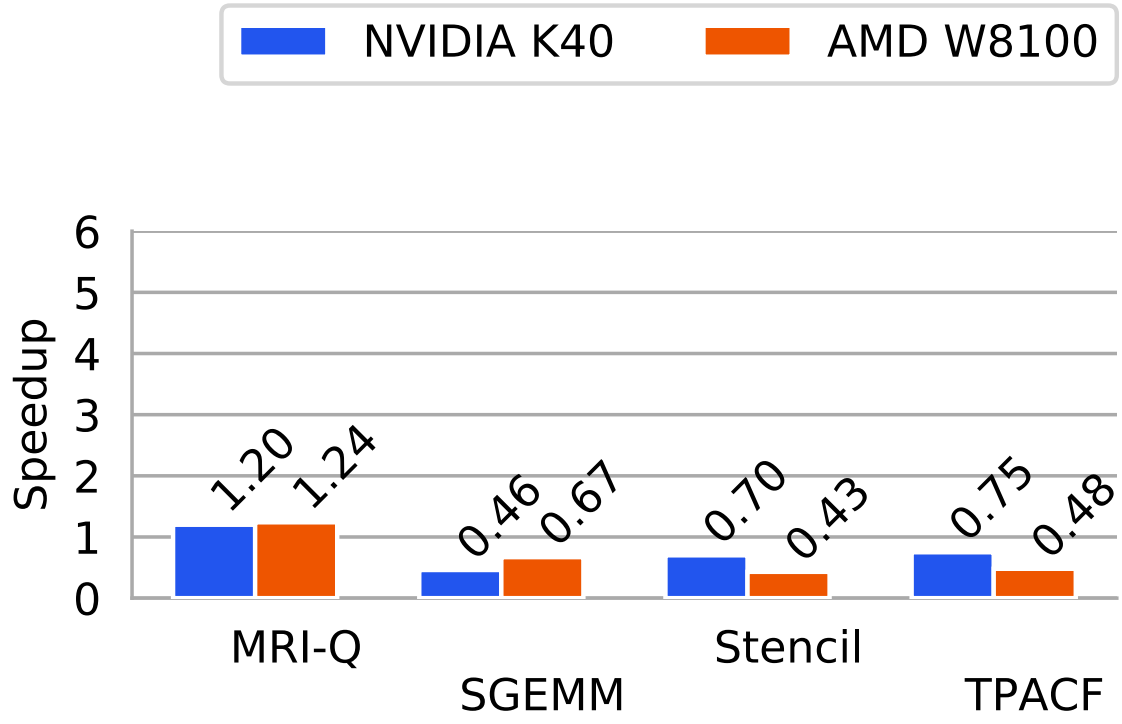
```
map (λx →
       loop ys=ys0 for i < n do
         map (λy → ...) ys)
     xs
```

Futhark maps this to GPU code by *interchanging* the outer **map** inwards:

```
loop yss=yss0 for i < n do
  map (λx ys →
         map (λy → ...) ys)
      xs yss
```

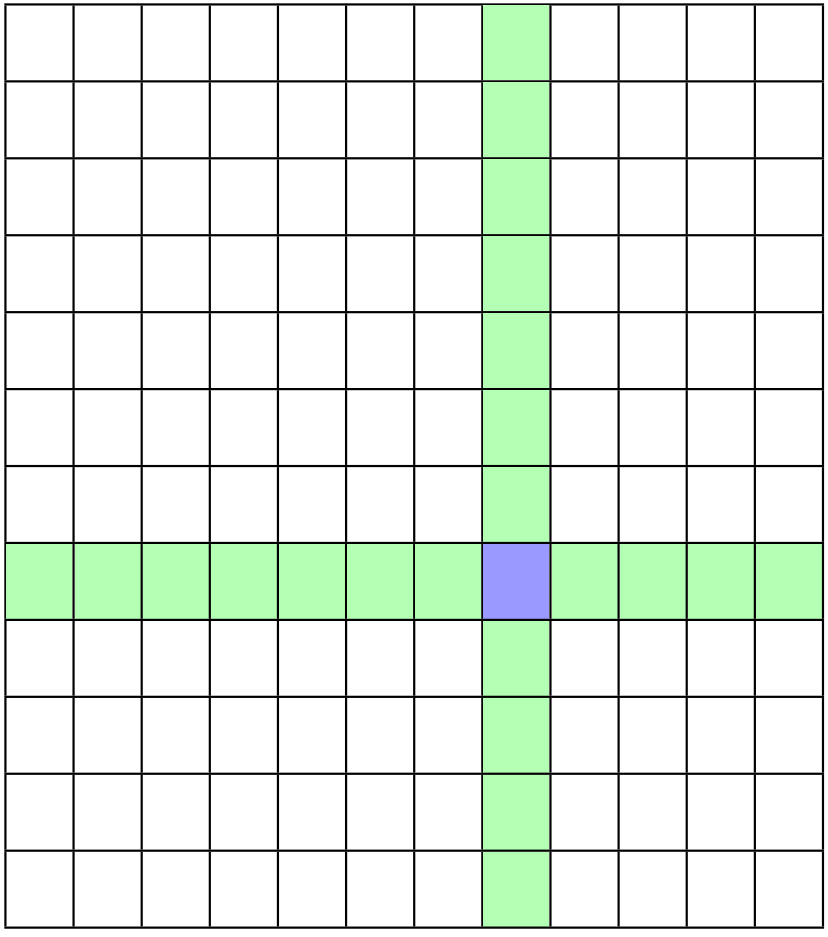
Results in n kernel launches, and n stores of intermediate results to global memory.
The Rodinia implementation executes the inner **map** directly in a GPU group, saving a *lot* of overhead.

Parboil



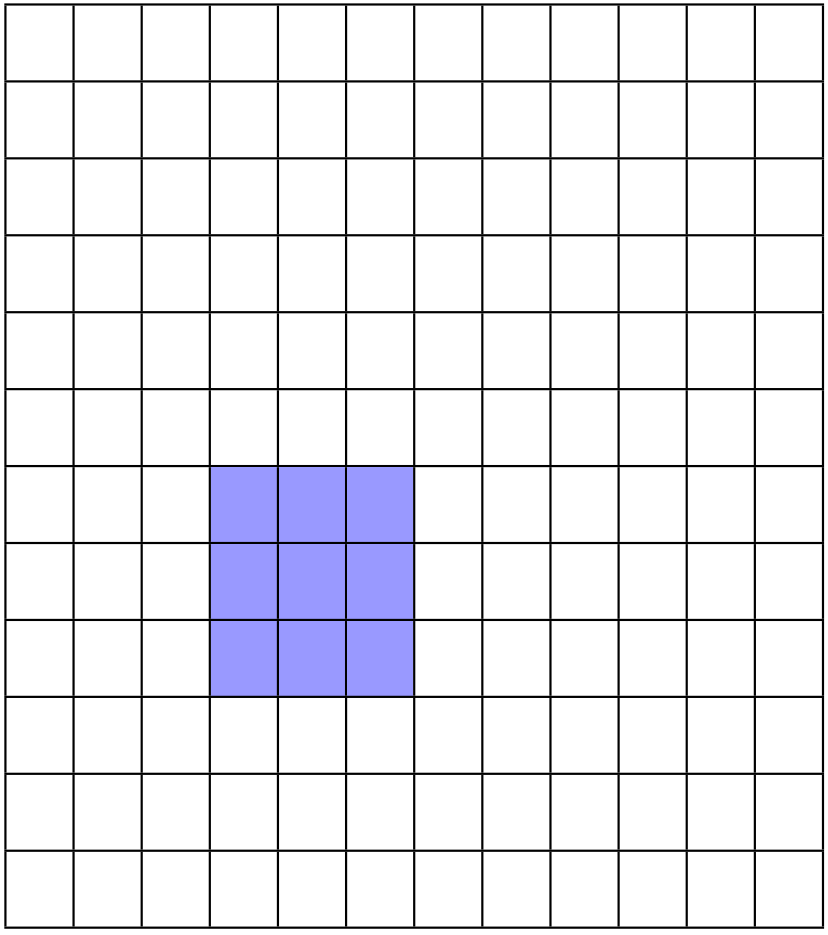
- CUDA and OpenCL implementations of consistently high quality.
- That we even get this close is a pretty good result!

Parboil: SGEMM — matrix multiplication
0.46× speedup



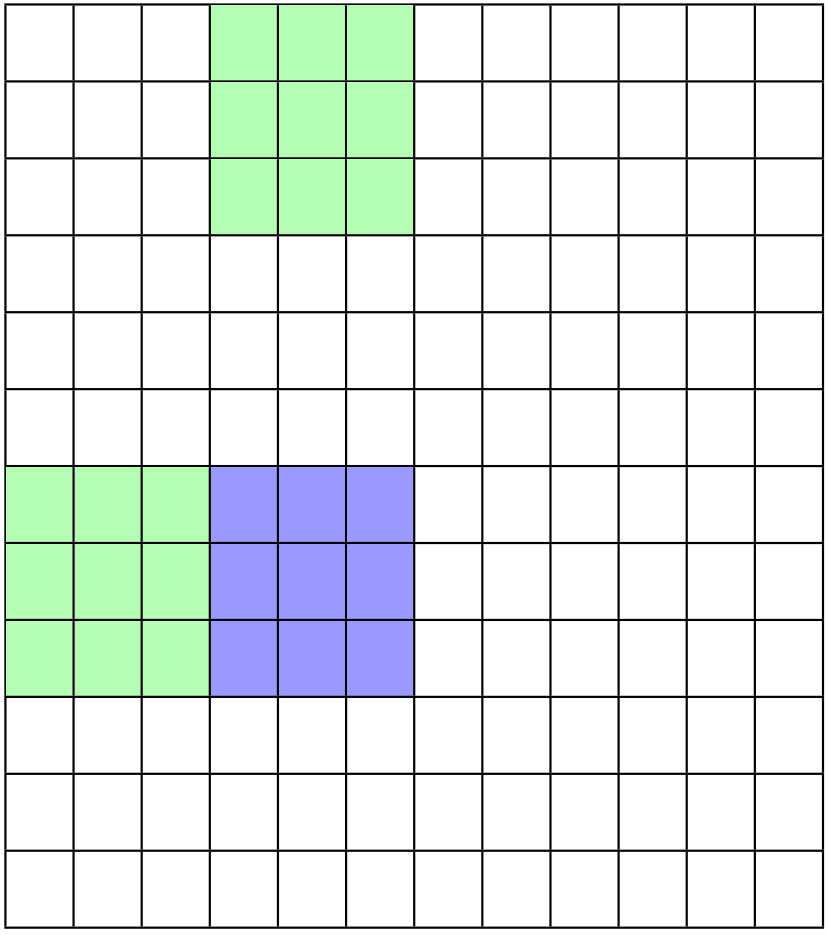
- One output element depends on a *row* and *column* of the two input matrices.
- Each output element can be computed in parallel. The inner dot product could as well, but not worth it in practice.
- We can improve the memory access pattern by *tiling*.

Parboil: SGEMM — applying tiling



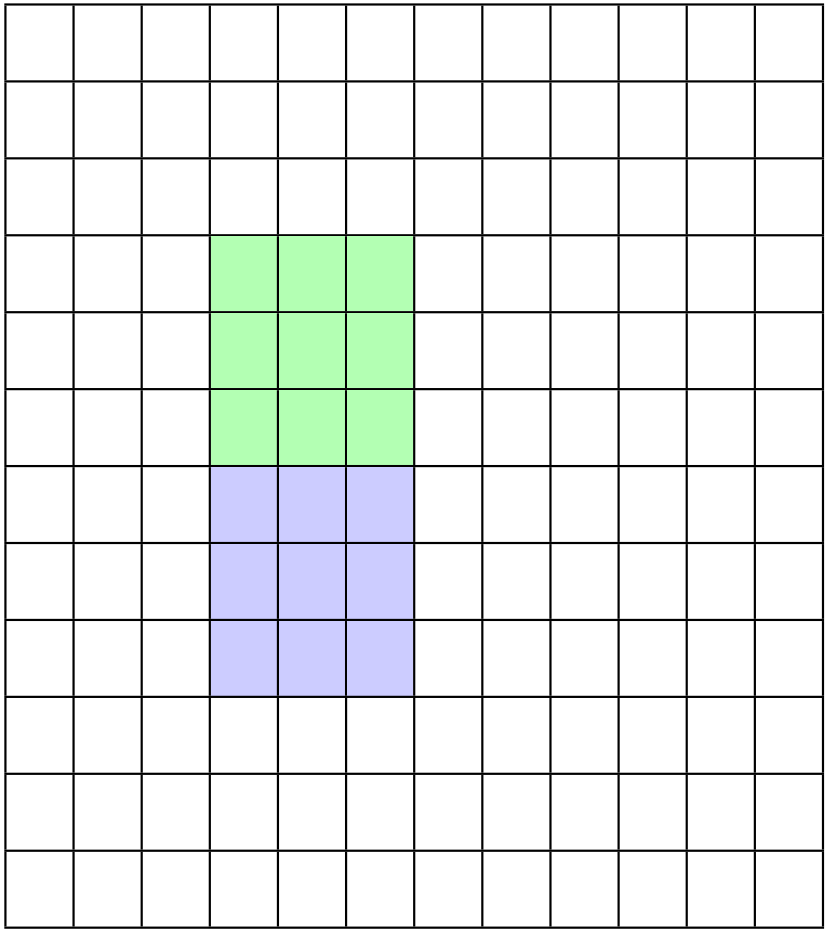
- Output matrix divided among GPU *groups*, still one thread per output element.
- Threads in a group collectively read two *tiles* into GPU local memory.
- Interim per-thread result is updated by looping across the tiles.

Parboil: SGEMM — applying tiling



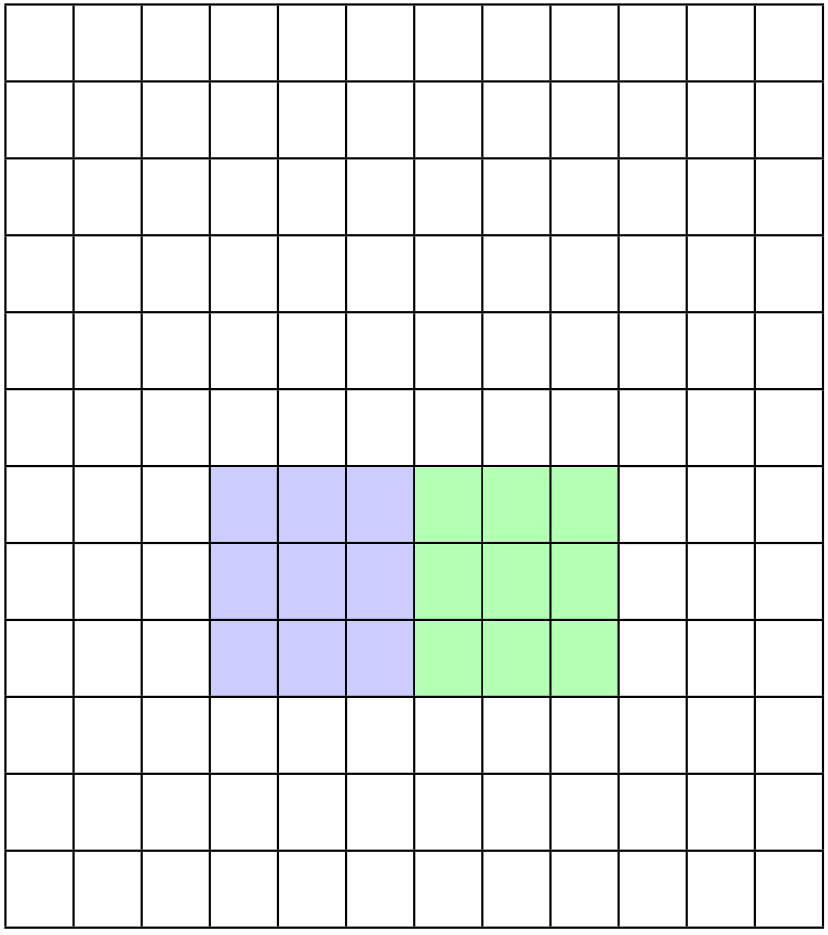
- Output matrix divided among GPU *groups*, still one thread per output element.
- Threads in a group collectively read two *tiles* into GPU local memory.
- Interim per-thread result is updated by looping across the tiles.

Parboil: SGEMM — applying tiling



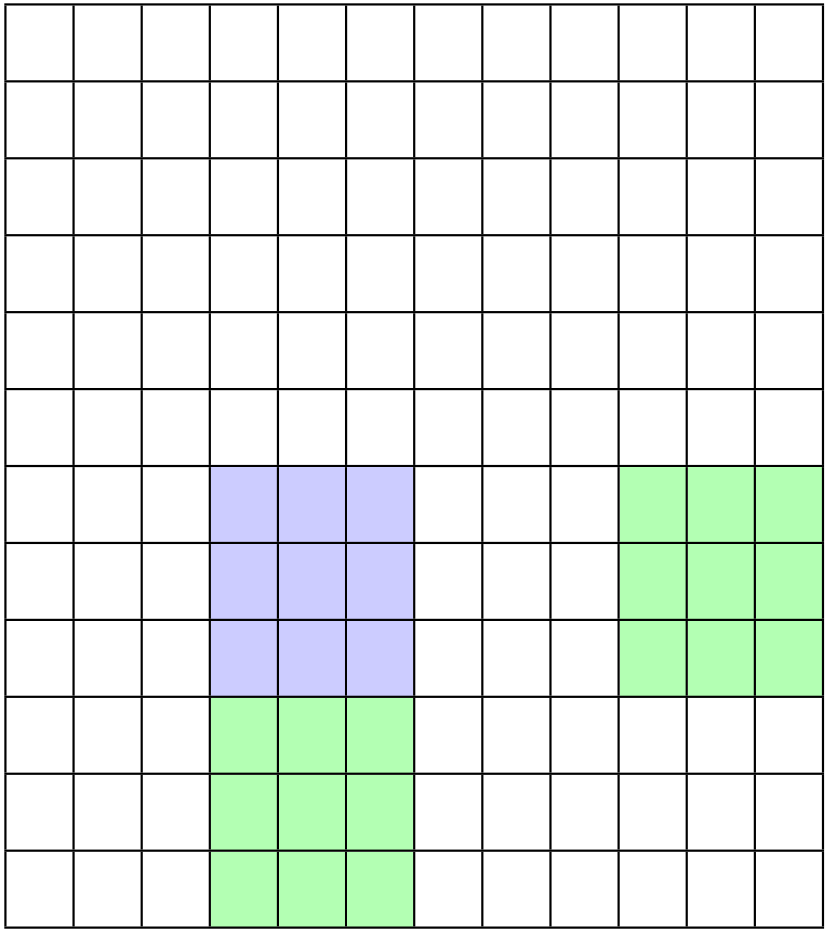
- Output matrix divided among GPU *groups*, still one thread per output element.
- Threads in a group collectively read two *tiles* into GPU local memory.
- Interim per-thread result is updated by looping across the tiles.

Parboil: SGEMM — applying tiling



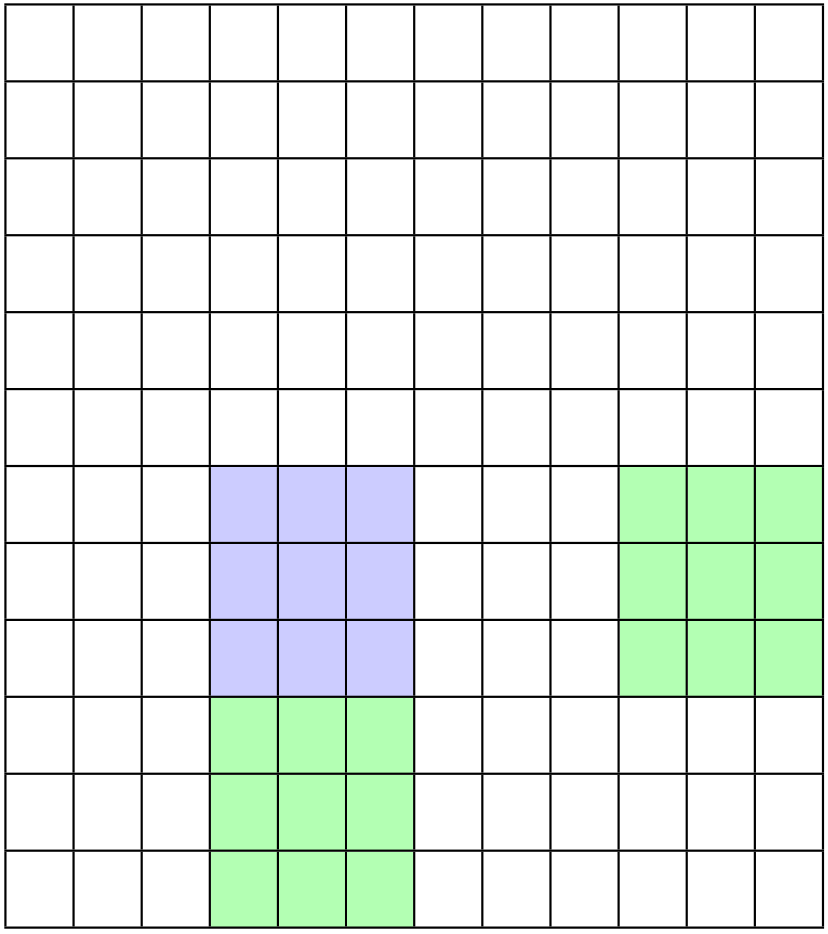
- Output matrix divided among GPU *groups*, still one thread per output element.
- Threads in a group collectively read two *tiles* into GPU local memory.
- Interim per-thread result is updated by looping across the tiles.

Parboil: SGEMM — applying tiling



- Output matrix divided among GPU *groups*, still one thread per output element.
- Threads in a group collectively read two *tiles* into GPU local memory.
- Interim per-thread result is updated by looping across the tiles.

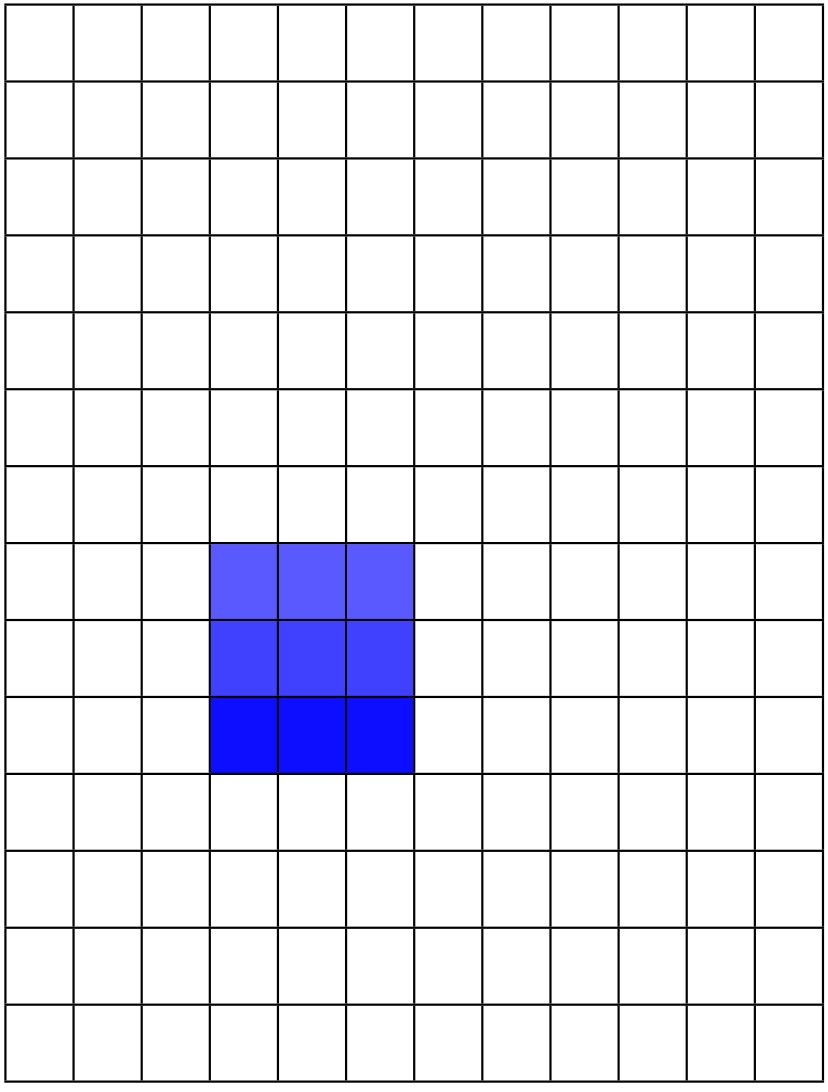
Parboil: SGEMM — applying tiling



- Output matrix divided among GPU *groups*, still one thread per output element.
- Threads in a group collectively read two *tiles* into GPU local memory.
- Interim per-thread result is updated by looping across the tiles.

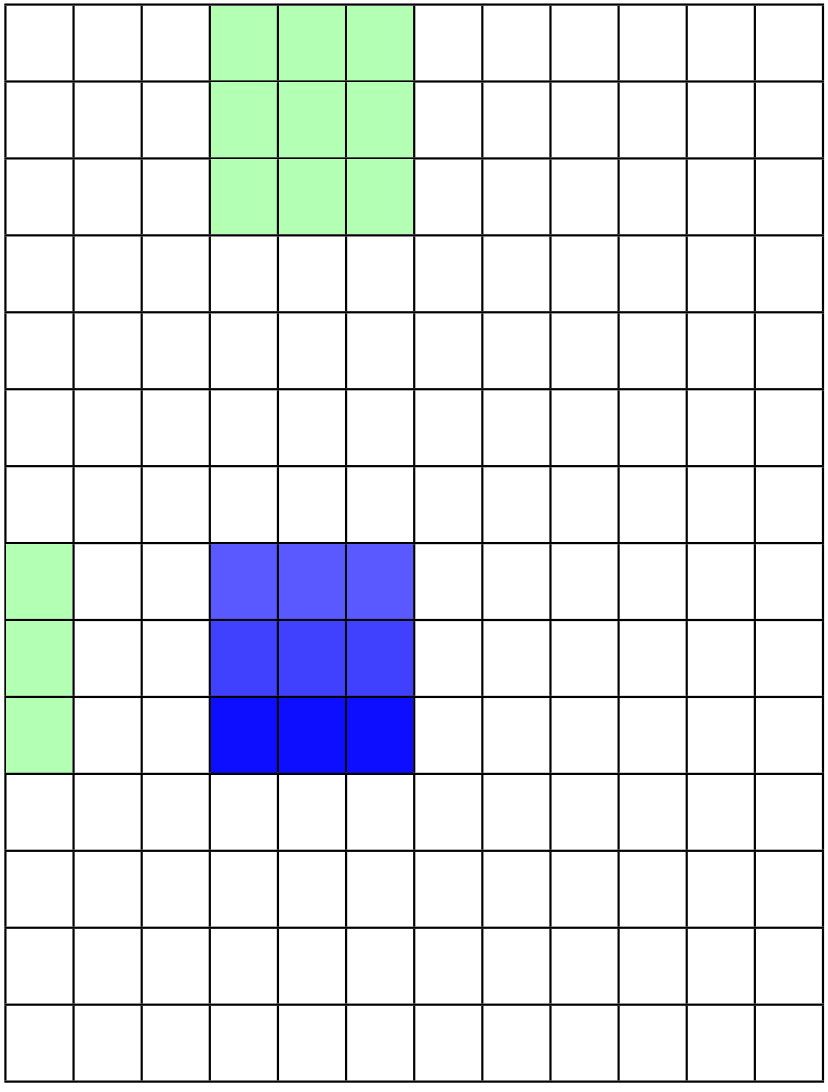
This is what Futhark does.

Parboil: SGEMM — register tiling



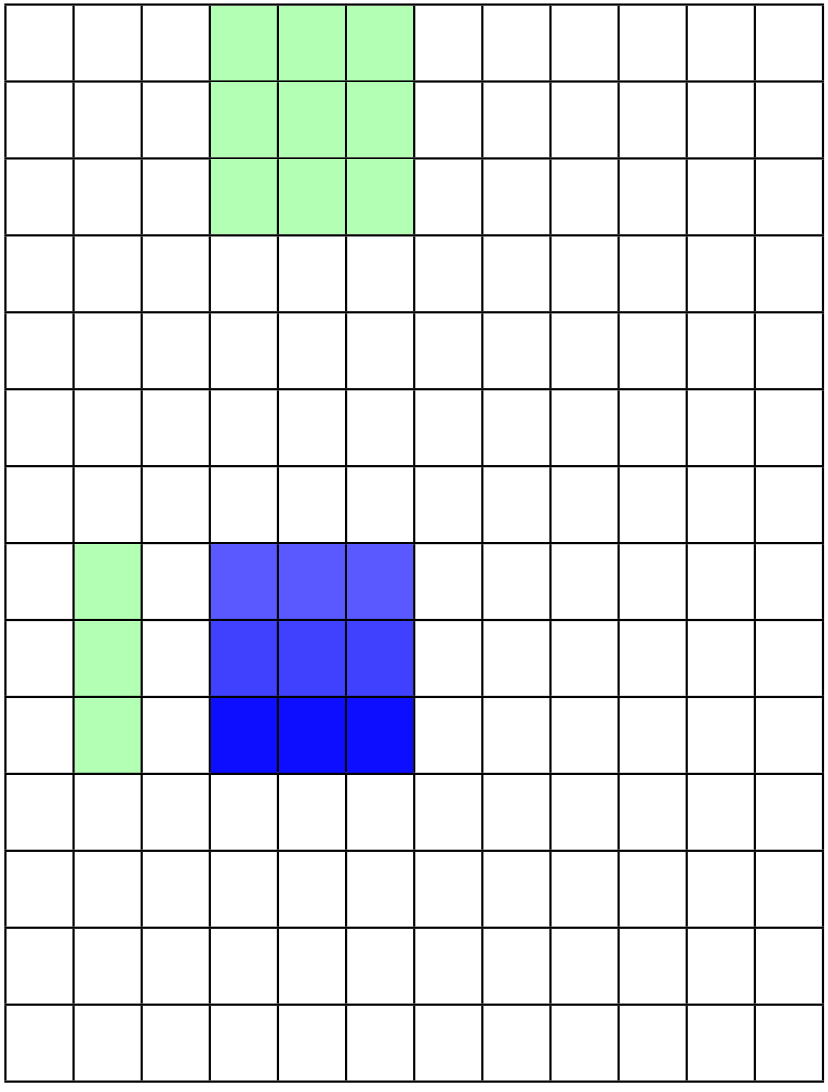
- Each thread responsible for *multiple* output elements — here 3, but 16 in Parboil — kept in registers until the end.
- Uses local-memory only in *one* direction.

Parboil: SGEMM — register tiling



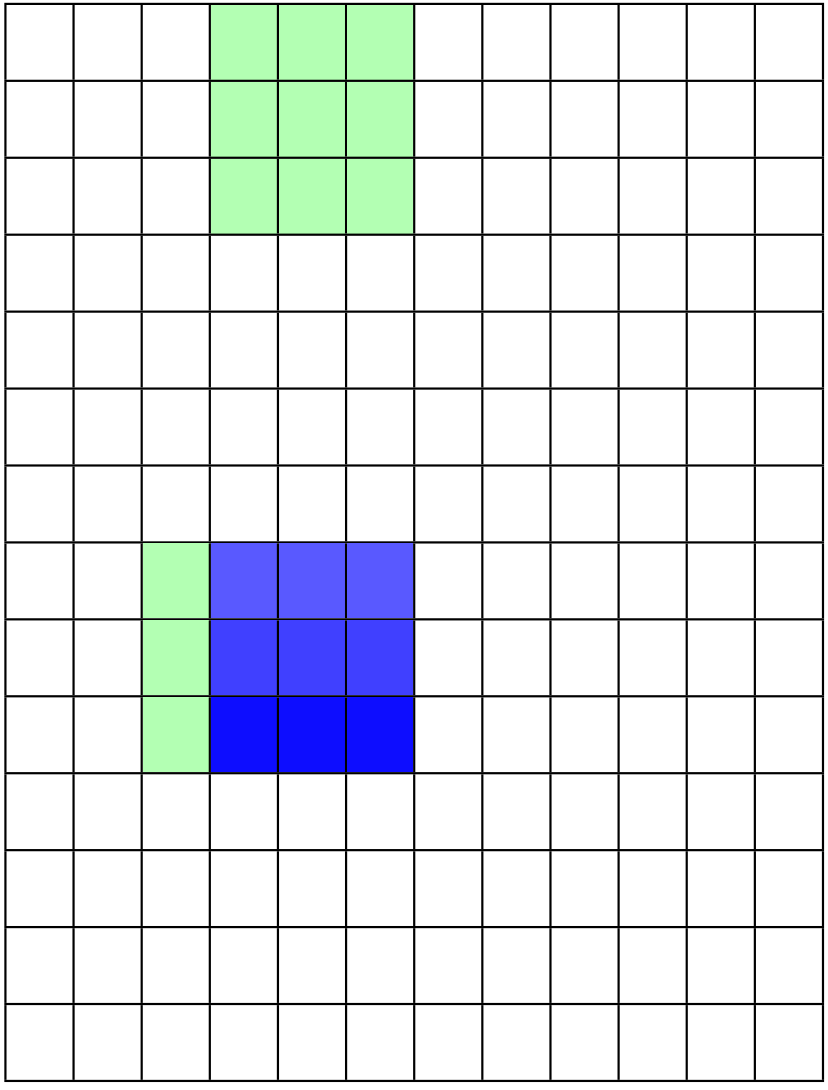
- Each thread responsible for *multiple* output elements — here 3, but 16 in Parboil — kept in registers until the end.
- Uses local-memory only in *one* direction.

Parboil: SGEMM — register tiling



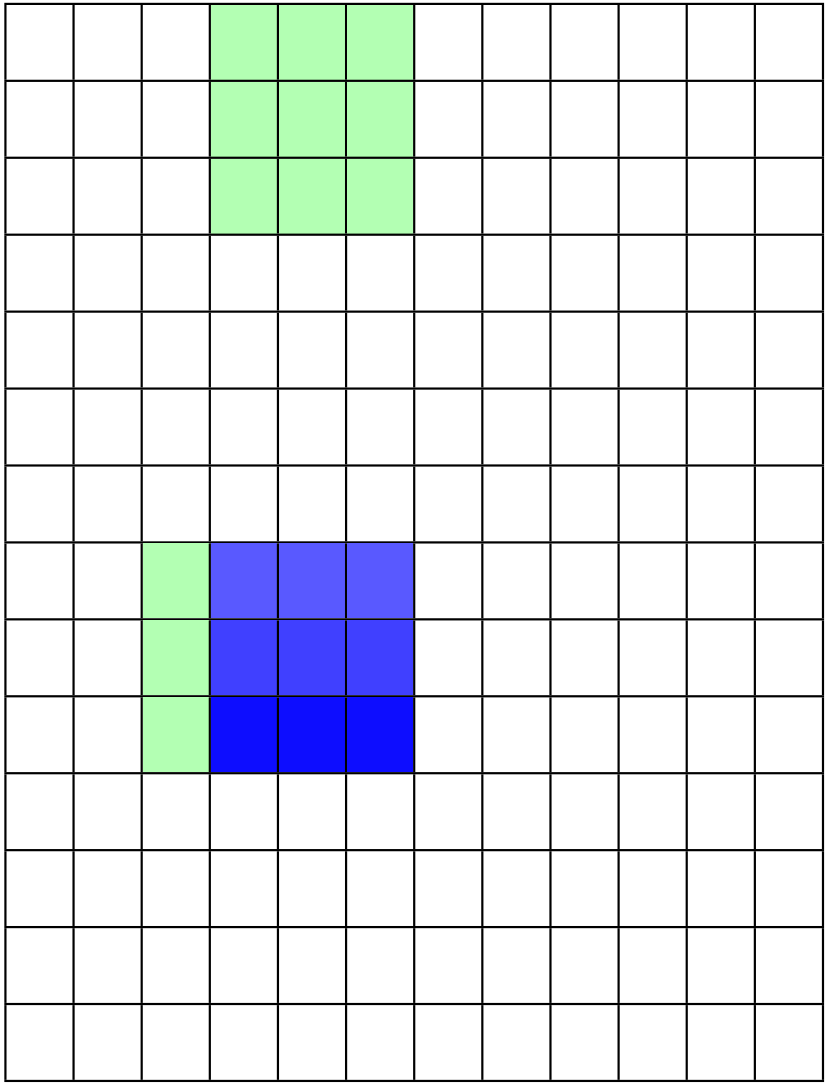
- Each thread responsible for *multiple* output elements — here 3, but 16 in Parboil — kept in registers until the end.
- Uses local-memory only in *one* direction.

Parboil: SGEMM — register tiling



- Each thread responsible for *multiple* output elements — here 3, but 16 in Parboil — kept in registers until the end.
- Uses local-memory only in *one* direction.

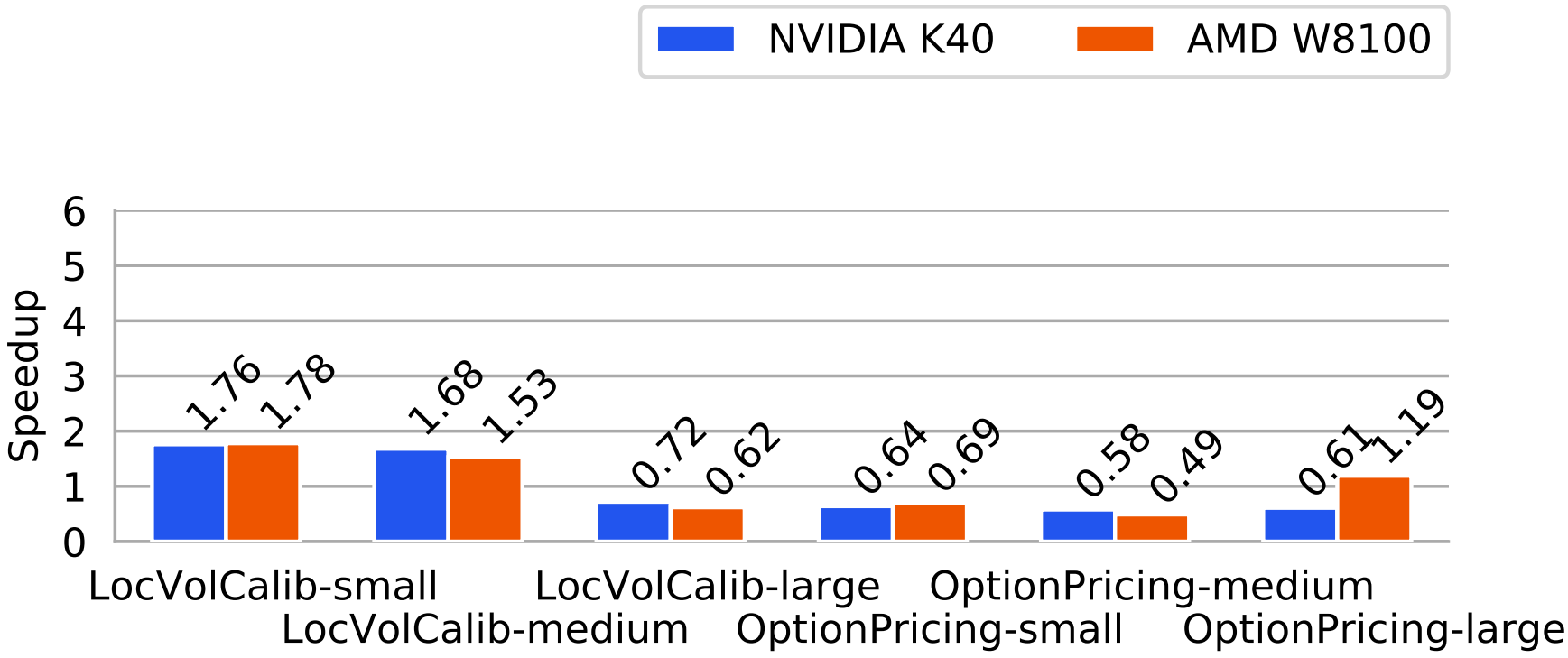
Parboil: SGEMM — register tiling



- Each thread responsible for *multiple* output elements — here 3, but 16 in Parboil — kept in registers until the end.
- Uses local-memory only in *one* direction.

Sacrifices parallelism to amortise the communication cost of tiling—this is why Parboil's SGEMM outperforms Futhark's.

FinPar



- OpenCL implementations of high quality (written by my advisor, so of course).
- Datasets have interesting properties.

FinPar: LocVolCalib
1.76×, 1.68×, 0.72× speedup

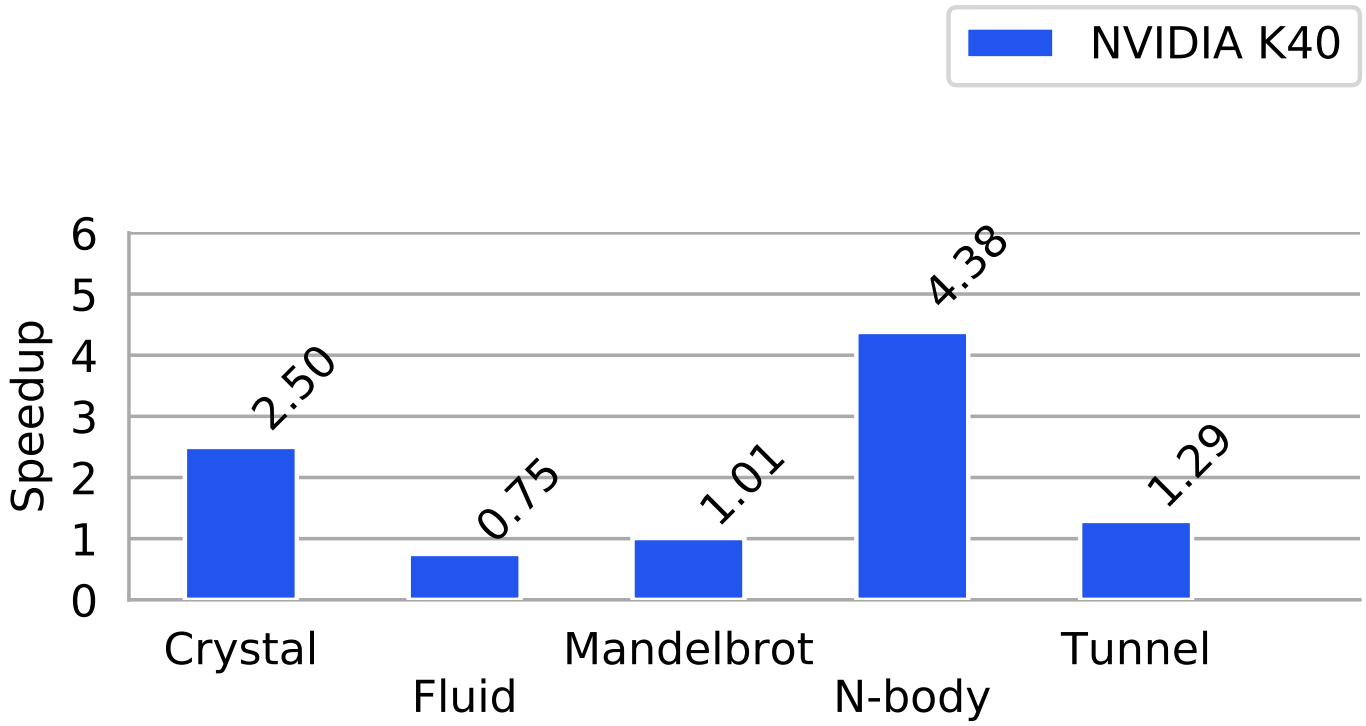
Core part contains an expression roughly like

```
map (λxs →
       let xs' = scan (+) 0 xs
       let xs'' = reduce (+) 0 xs'
       let xs''' = map (f xs'') xs'
       in xs)
    xss
```

- Should the inner parallism be exploited? Depends on size of xss.
- Smaller two datasets do not have enough *parallelism* in the outer **map** to saturate the GPU.
- Both Futhark and reference implementations focused on the largest dataset.

Future solution: multi-versioned code.

Accelerate



- High-level parallel array language embedded in Haskell.
- Only supports NVIDIA GPUs.
- Benchmarks generally pretty simple, with no nested parallelism, as it is not supported by Accelerate.

Accelerate: N-body
4.38× speedup

```
map (λbody → reduce sum_accel 0 zero_accel
                    (map (acceleration body) bodies))
    bodies
```

- Inner **map**–**reduce** is sequentialised.
- Each thread is looping over the same array: bodies.

Accelerate: N-body
4.38× speedup

```
map (λbody → reduce sum_accel 0 zero_accel
                (map (acceleration body) bodies))
    bodies
```

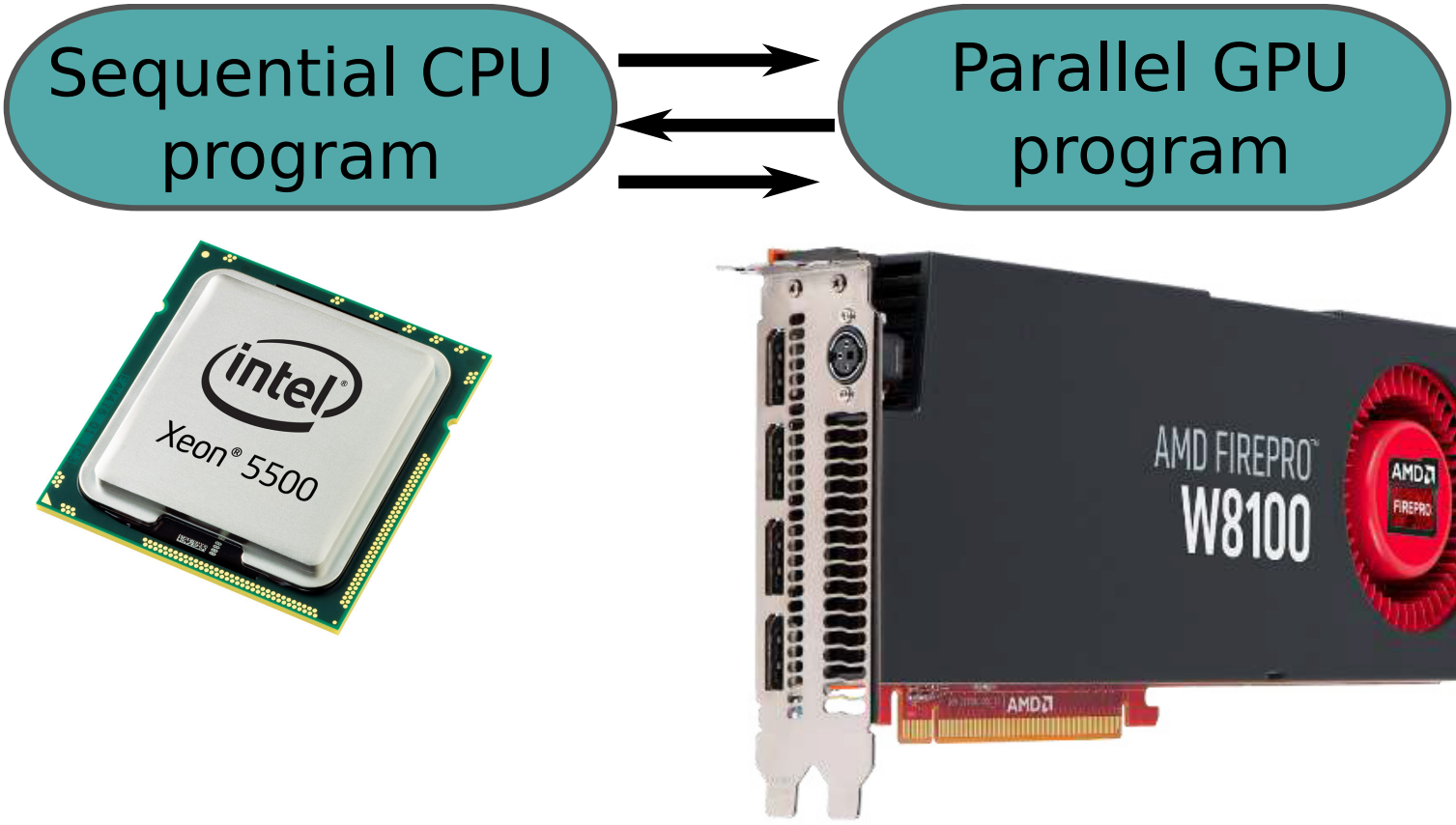
- Inner **map**–**reduce** is sequentialised.
- Each thread is looping over the same array: bodies.
- This can be tiled in local memory!
- This gives the speedup over accelerate.

```
map (λbody →
       stream_seq (λbodies' →
                      reduce sum_accel 0 zero_accel
                      (map (acceleration body) bodies'))
                  bodies)
    bodies
```

So it's fast, but is it usable?

We can give indirect proof of usability by implementing real(-ish) software in Futhark.

The Enabling Division



The controlling CPU program does *not* have to be fast. It can be generated in a language that is *convenient*.

Compiling Futhark to Python+PyOpenCL

```
entry sum_nats (n: i32): i32 =
  reduce (+) 0 [1...n]

$ futhark-pyopencl --library sum.fut
```

This creates a Python module `sum.py` which we can use as follows:

```
$ python
>>> from sum import sum
>>> c = sum()
>>> c.sum_nats(10)
55
>>> c.sum_nats(1000000)
1784293664
```

Good choice for all your integer summation needs!

Compiling Futhark to Python+PyOpenCL

```
entry sum_nats (n: i32): i32 =
  reduce (+) 0 [1...n]

$ futhark-pyopencl --library sum.fut
```

This creates a Python module `sum.py` which we can use as follows:

```
$ python
>>> from sum import sum
>>> c = sum()
>>> c.sum_nats(10)
55
>>> c.sum_nats(1000000)
1784293664
```

Good choice for all your integer summation needs!

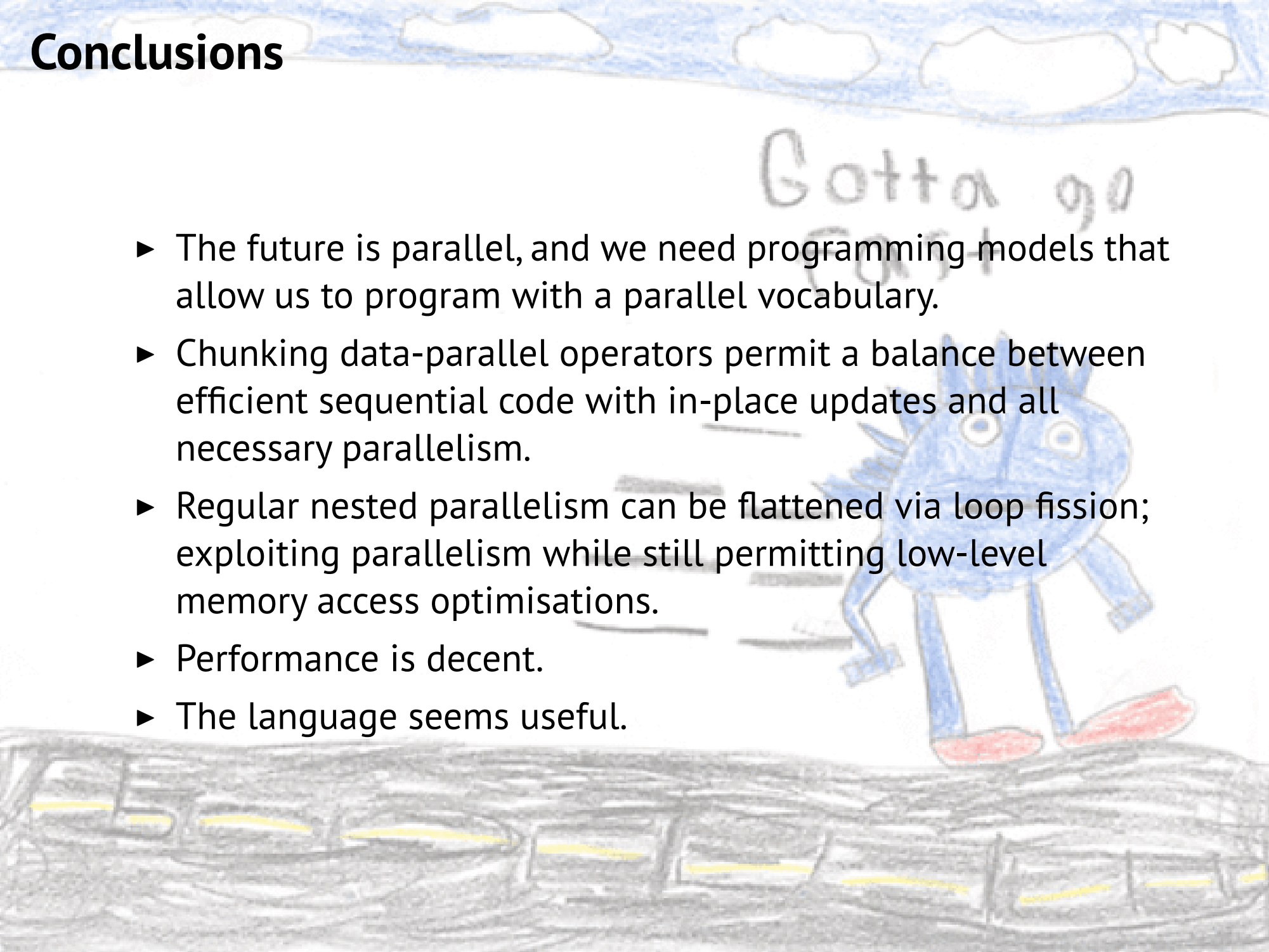
Or, we could have our Futhark program return an array containing pixel colour values, and use Pygame to blit it to the screen...

CONCLUSIONS

Well, that wasn't so hard!

—Sonic the Hedgehog (Sonic Adventure, 1998)

Conclusions



- The future is parallel, and we need programming models that allow us to program with a parallel vocabulary.
- 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; exploiting parallelism while still permitting low-level memory access optimisations.
- Performance is decent.
- The language seems useful.

APPENDICES

Keep it up. There are still lots more left!
— Dr. Eggman (Shadow the Hedgehog, 2001)

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
```

Why GPUs?

Cheap Easing their programming has a democratising effect, as it makes high-performance computation more accessible.

Widespread This room contains dozens of GPUs, but likely not even a single *programmable* FPGA (unless you brought one).

Restricted If Futhark can be translated to GPU code, then it can likely also be translated to other platforms.

“**Easy**” The performance guidelines for GPUs are fairly straightforward. Applying them manually is extremely tedious and error-prone, but they are a good fit for a compiler.

Future Directions

Dataset-sensitive compilation

There is no *one size fits all* for parallelisation. The compiler should generate multiple program versions, and pick the optimal one at runtime based on the concrete dataset.

Flexibility

Support for irregular problems, such as graph algorithms. Not *at all costs*—other more expressive languages already exist. Futhark's gotta go fast.

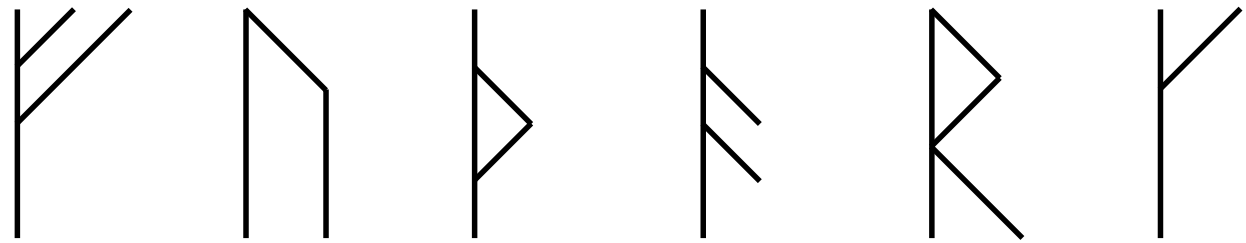
Clusters

Single computers are so 2000—transparent distributed computing on moderately sized clusters would be useful.

Heterogeneity

Splitting computation between both CPU and GPU.

Futhark?



f u th a r k

(With thanks to Torben Mogensen.)