Using smoke and mirrors to compile a functional programming language to efficient GPU code

Troels Henriksen (athas@sigkill.dk), University of Copenhagen

lambda DAYS

28-29 JULY 2022

KRAKÓW | POLAND

- Troels Henriksen.
- Assistant professor researcher at the Department of Computer Science at the University of Copenhagen (DIKU).
- All this is joint work with **Cosmin Oancea**, **Philip Munksgaard**, **Robert Schenck**, **Martin Elsman**, and various open source contributors way.

When I was first taught functional programming, I was told it would be the future because it makes parallel execution trivial.

E.g. `f(g(x),h(y))` — in a pure language, `g(x)` and `h(y)` can be executed in parallel.

When I was first taught functional programming, I was told it would be the future because it makes parallel execution trivial.

E.g. `f(g(x),h(y))` — in a pure language, `g(x)` and `h(y)` can be executed in parallel.

It's the future *now*—where's all the parallel functional programming?

Why hasn’t parallel functional programming taken over the world?

Why hasn't parallel functional programming taken over the world?

- **Counterclaim: it has!** Lots of parallel and concurrent programming libraries baed on functional concepts:
    - Akka (Scala), TensorFlow (Python), Accelerate (Haskell)...

Why hasn’t parallel functional programming taken over the world?

- **Counterclaim: it has!** Lots of parallel and concurrent programming libraries baed on functional concepts:
  - Akka (Scala), TensorFlow (Python), Accelerate (Haskell)…
- **But that’s not how I understood it!**
  - GPUs are modern parallel computers…

Why hasn’t parallel functional programming taken over the world?

- **Counterclaim: it has!** Lots of parallel and concurrent programming libraries baed on functional concepts:
  - Akka (Scala), TensorFlow (Python), Accelerate (Haskell)...
- **But that’s not how I understood it!**
  - GPUs are modern parallel computers...
  - **...so why can’t my compiler automatically turn my Scala/Haskell/OCaml program into e.g. fast GPU code?**

Problems and potential

- **Parallelising small nuggets of work is not efficient on current computers.**
  - `f(g(x),h(y))`

Problems and potential

- **Parallelising small nuggets of work is not efficient on current computers.**
  - `f(g(x),h(y))`
- Some people work on hardware designed for functional programming, but I want to use **existing, consumer parallel hardware, such as GPUs.**

The big question

How do we go from *idiomatic functional code* to the kind of low-level programming style expected by a GPU?

Idiomatic code

- Some kinds of functional programming are not suited for GPU parallelisation.
  - E.g. tiny recursive steps over sequential data structures like lists.
- But **bulk data transformations** with higher order functions is *very well suited*!
  - `map`
  - `reduce`
  - `scan`
  - `filter`
  - ...

(Incidentally, that kind of style is also what most high level parallel libraries are designed for.)

This is how I want to write parallel programs

```
def dotprod [n] (x: [n]f32) (y: [n]f32) =
  f32.sum (map2 (*) x y)
```

This is how I want to write parallel programs

```
def dotprod [n] (x: [n]f32) (y: [n]f32) =
  f32.sum (map2 (*) x y)

def matmul [n][m][k] (A: [n][m]f32) (B: [m][k]f32) =
  map (\A_row -> map (\B_col -> dotprod A_row B_col)
                     (transpose B))
      A
```

This is how I want to write parallel programs

```
def dotprod [n] (x: [n]f32) (y: [n]f32) =
  f32.sum (map2 (*) x y)

def matmul [n][m][k] (A: [n][m]f32) (B: [m][k]f32) =
  map (\A_row -> map (\B_col -> dotprod A_row B_col)
                     (transpose B))
      A
```

- This is **Futhark**—a *small* parallel functional language in the ML tradition that we develop at DIKU.
- Compiles to GPU or CPU code.
- **By design very much a “least common denominator” language.**

Let’s talk about GPUs

- A GPU function is called a **kernel**.
    - Typically consists of *tens of thousands of threads*.
    - All threads run *the same code*.

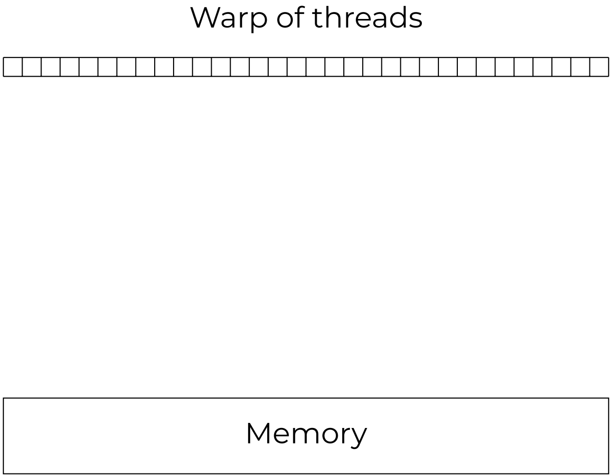
Let’s talk about GPUs

- A GPU function is called a **kernel**.
  - Typically consists of *tens of thousands of threads*.
  - All threads run *the same code*.

```
kernel (int* arr) {
  var i = get_thread_id();
  var x = arr[i];
  if (x < 0) {
    arr[i] = -x;
  }
}
```

- Threads are split into **warps**, which execute in **lockstep**.
- **Regularity** is important.
- **Memory access** usually the bottleneck.

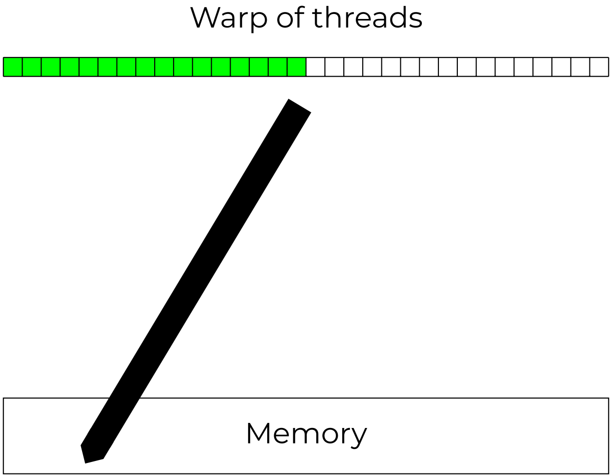
Coalesced memory accesses



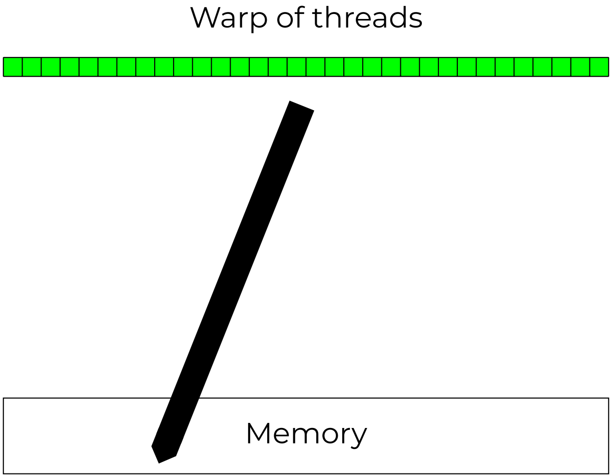
Warp of threads

Memory

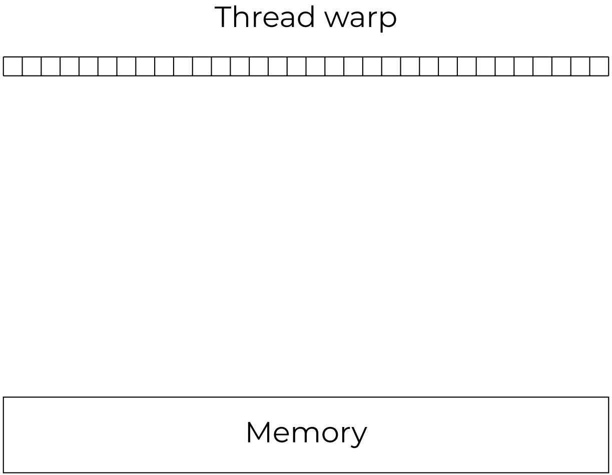
Coalesced memory accesses



Coalesced memory accesses



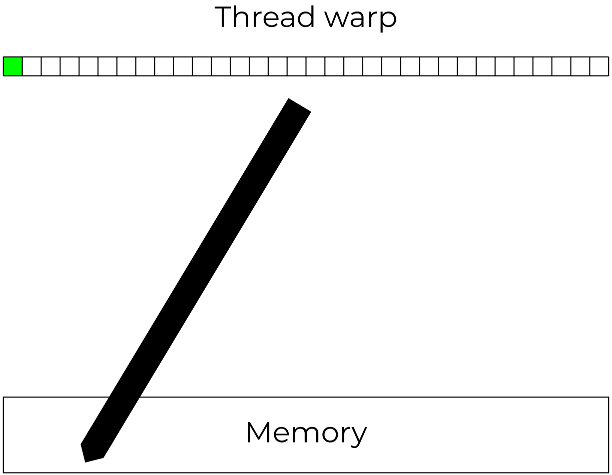
Uncoalesced memory accesses



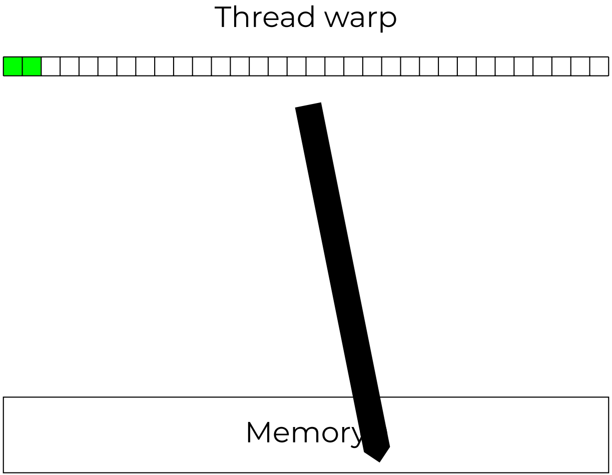
Thread warp

Memory

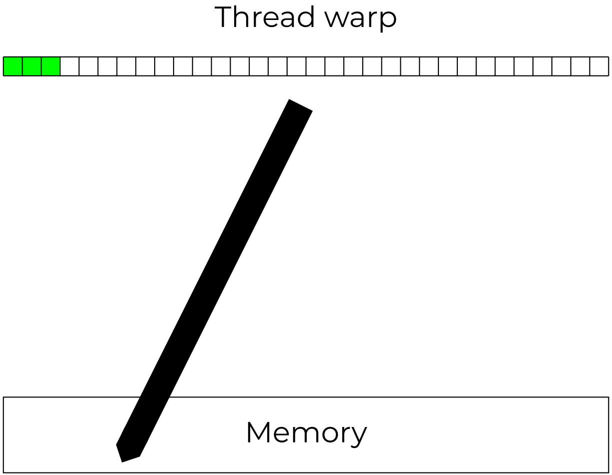
Uncoalesced memory accesses



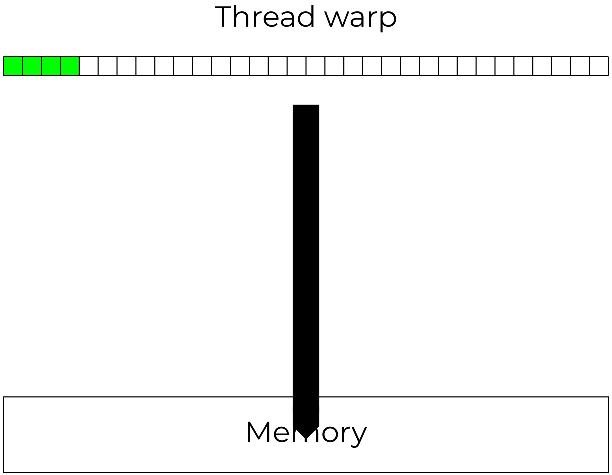
Uncoalesced memory accesses



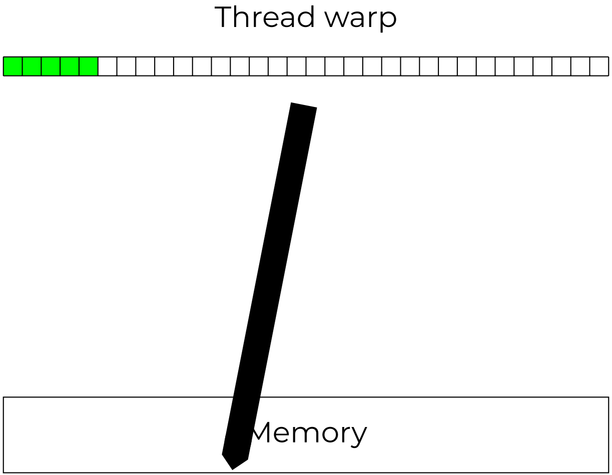
Uncoalesced memory accesses



Uncoalesced memory accesses



Uncoalesced memory accesses



Compiling a functional language to GPU

This is difficult!

Compiling a functional language to GPU

This is difficult!

This is not yet fully solved!

Compiling a functional language to GPU

This is difficult!

This is not yet fully solved!

The trick

Solve an easier problem by removing some language features and hope programmers won't notice.

Let’s talk about value representation

Let’s talk about value representation

Most fundamental principle

Futhark **unboxes** all non-arrays to keep them in registers.

- A triple `(a,b,c)` is treated as three distinct values.
- A record `{x:  f32, y:  f32}` is syntactic sugar for a tuple `(f32,f32)`

Monomorphisation

```
def swap ’a ’b (x: a, y: b) = (y,x)
```

Monomorphisation

```
def swap ’a ’b (x: a, y: b) = (y,x)

… swap (1,true)…
```

Monomorphisation

```
def swap ’a ’b (x: a, y: b) = (y,x)

… swap (1,true)…

def swap_i32_bool (x: bool, y: i32) = (y,x)

… swap_i32_bool (1,true)…
```

Let’s talk about arrays

Let’s talk about arrays

A = [[1, 2, 3, 4, 5, 6],
[7, 8, 9, 10, 11, 12],
[13, 14, 15, 16, 17, 18],
[19, 20, 21, 22, 23, 24]]

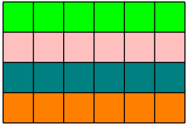
Let’s talk about arrays

A = [[1, 2, 3, 4, 5, 6],
[7, 8, 9, 10, 11, 12],
[13, 14, 15, 16, 17, 18],
[19, 20, 21, 22, 23, 24]]

```
map (foldl (+) 0) A
```

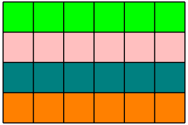
Array-of-arrays layout

A:

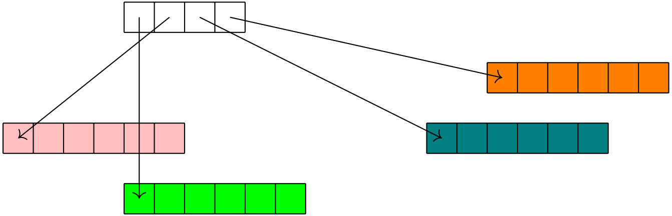


Array-of-arrays layout

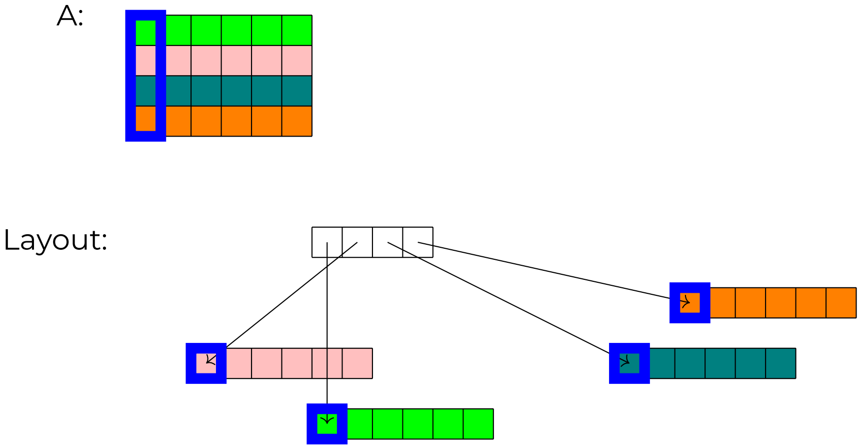
A:



Layout:



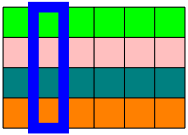
Array-of-arrays layout



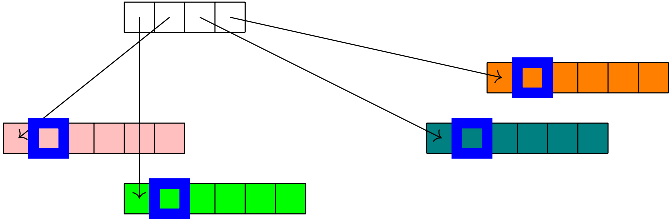
Uncoalesced memory accesses!

Array-of-arrays layout

A:

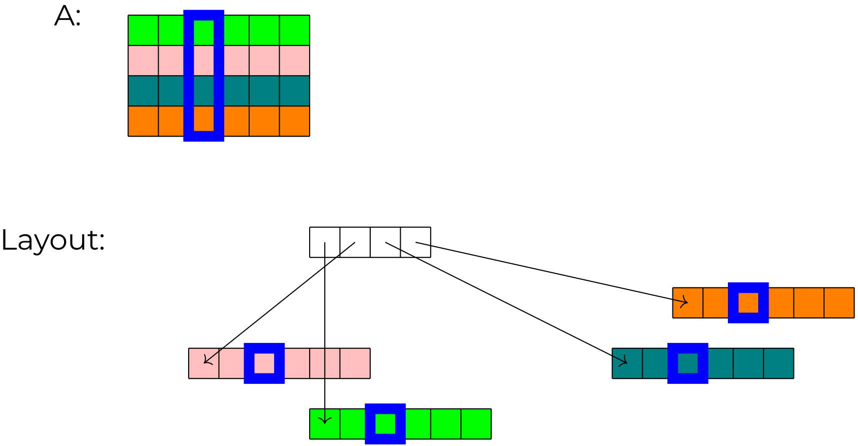


Layout:



Uncoalesced memory accesses!

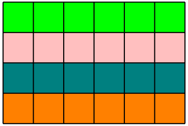
Array-of-arrays layout



Uncoalesced memory accesses!

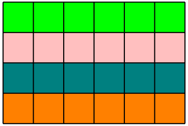
Dense row-major layout

A:

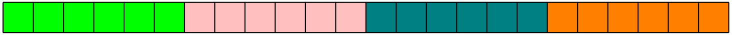


Dense row-major layout

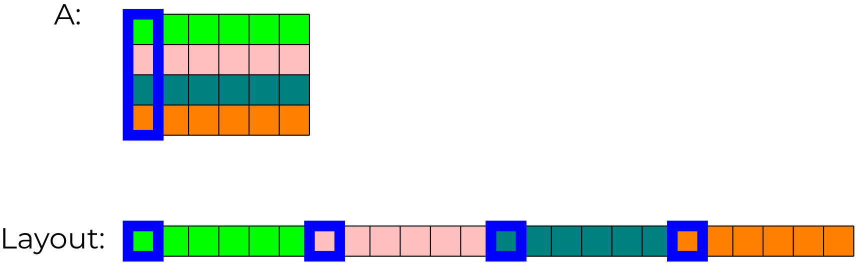
A:



Layout:

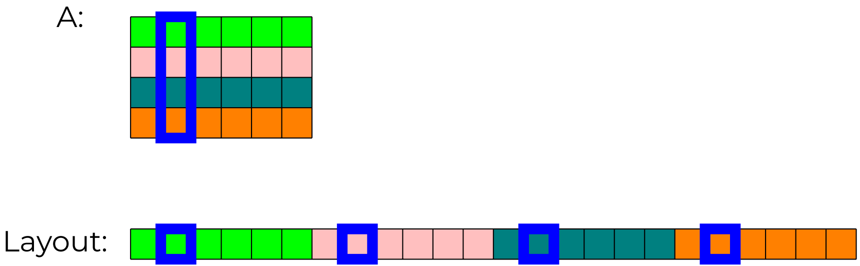


Dense row-major layout



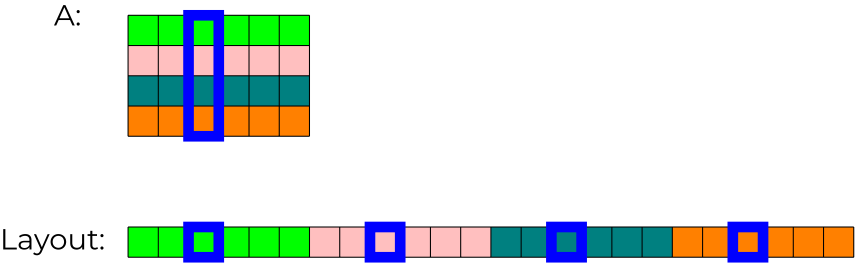
Uncoalesced memory accesses!

Dense row-major layout



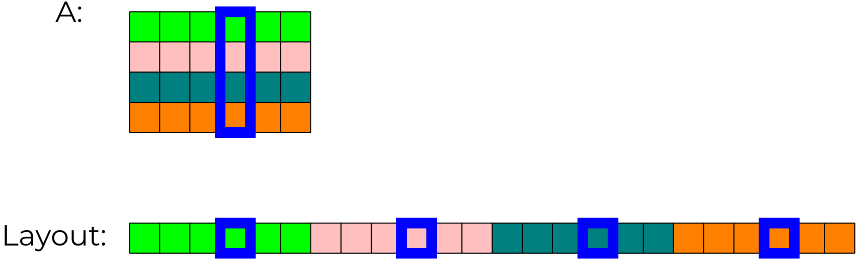
Uncoalesced memory accesses!

Dense row-major layout



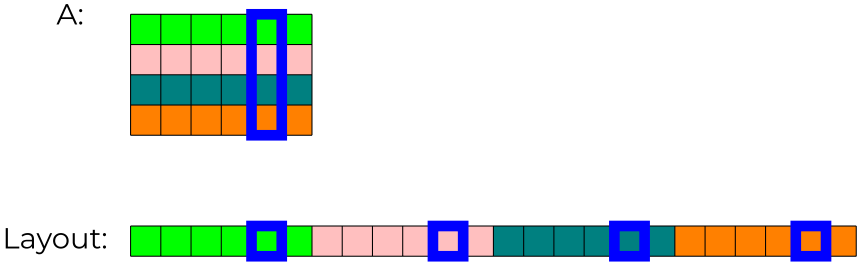
Uncoalesced memory accesses!

Dense row-major layout



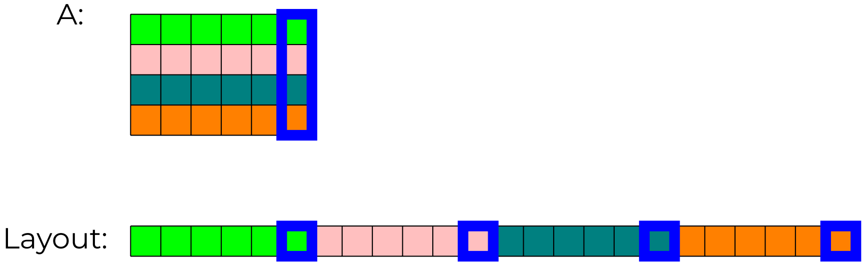
Uncoalesced memory accesses!

Dense row-major layout



Uncoalesced memory accesses!

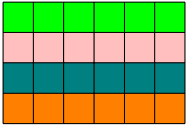
Dense row-major layout



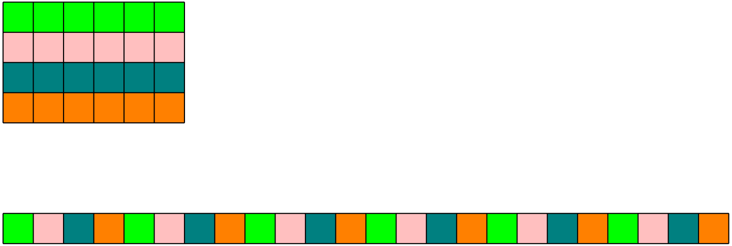
Uncoalesced memory accesses!

Dense column-major layout

A:

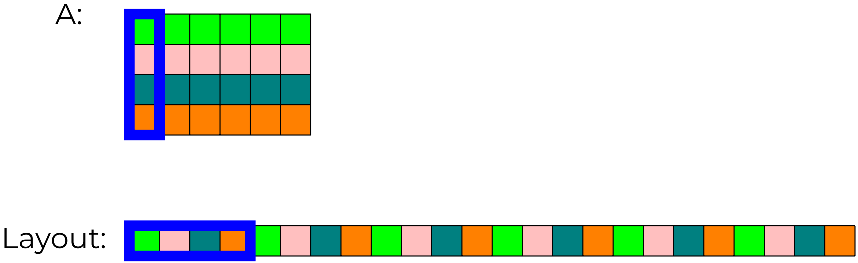


Dense column-major layout


A:

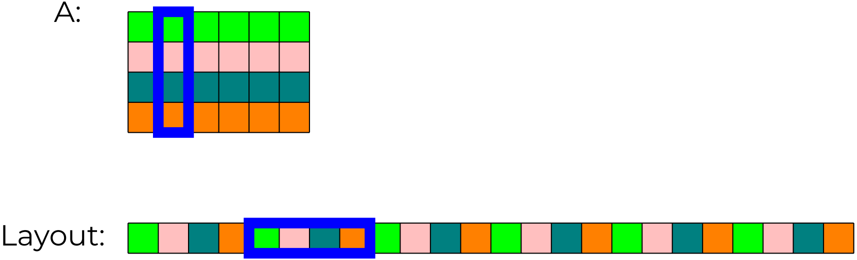
Layout:

Dense column-major layout



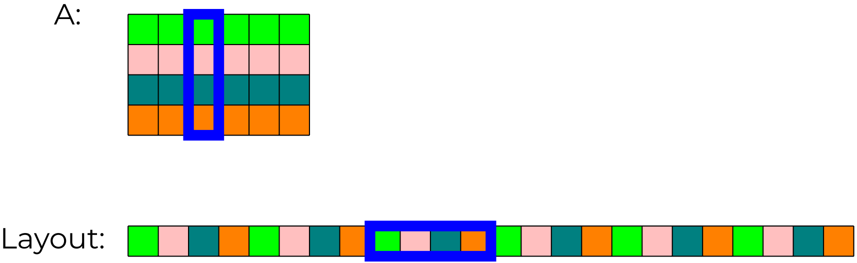
Coalesced memory accesses!

Dense column-major layout



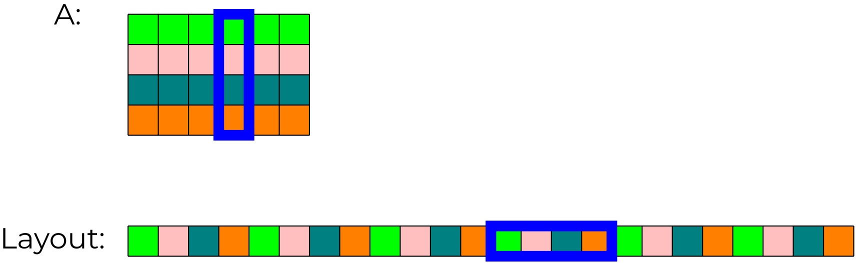
Coalesced memory accesses!

Dense column-major layout



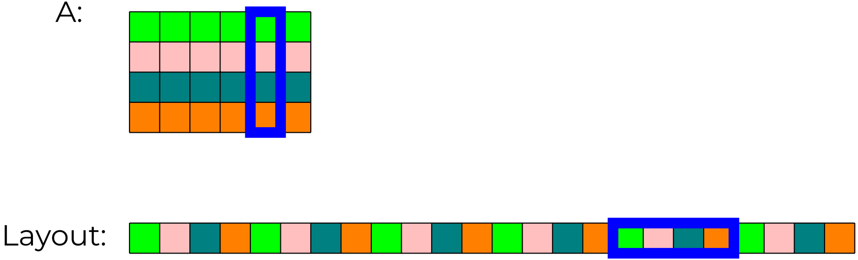
Coalesced memory accesses!

Dense column-major layout



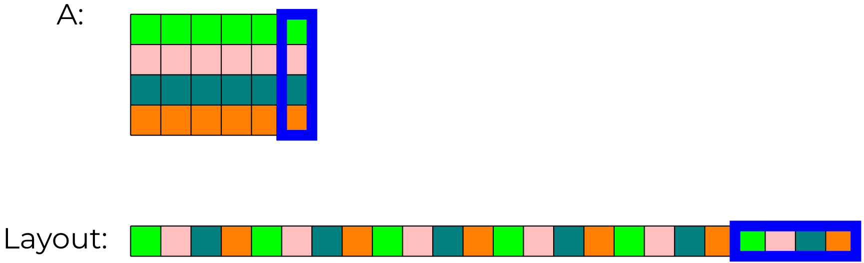
Coalesced memory accesses!

Dense column-major layout



Coalesced memory accesses!

Dense column-major layout



Coalesced memory accesses!

Illusion and reality

- We provide a **programming model** based on “arrays of arrays”.
- But in-memory representation is dense, in some layout decided on compiler.

Illusion and reality

- We provide a **programming model** based on “arrays of arrays”.
- But in-memory representation is dense, in some layout decided on compiler.
- **Key restriction:** arrays must be *regular*.

```
[[1,2,3], [4,5]] -- Forbidden!
```

Verified by the type checker using a **size-dependent type system** .

But it’s not just about multidimensional arrays

- Suppose we have *n* threads and they each sequential construct an array with *m* elements.

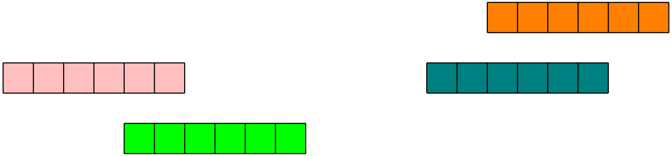
```
map (\x -> ...
           let b : [m]i32 = ...
           ...
    ) xs
```

But it’s not just about multidimensional arrays

- Suppose we have *n* threads and they each sequential construct an array with *m* elements.

```
map (\x -> ...
           let b : [m]i32 = ...
           ...
    ) xs
```

If each thread is given its own memory block, then we’re back to chasing pointers and uncoalesced memory.



Allocating ahead of time

```
let mem = alloc(n*m*sizeof(i32))
map (\x -> ...
           -- store all the 'b's in 'mem'
           let b : [m]i32 = ...
           ...
    ) xs
```

Allocating ahead of time

```
let mem = alloc(n*m*sizeof(i32))
map (\x -> ...
           -- store all the ’b’s in ’mem’
           let b : [m]i32 = ...
           ...
    ) xs
```

And we store arrays from different threads interleaved, to get coalesced access.

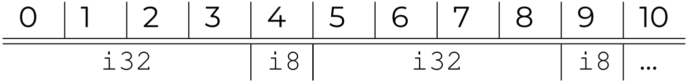
Arrays of tuples

Arrays of tuples

Consider arrays of type `[](i32, i8)`. Since an `i32` is four bytes and a `i8` is one byte, how should Futhark store this in memory?

Arrays of tuples

Consider arrays of type `[](i32, i8)`. Since an `i32` is four bytes and a `i8` is one byte, how should Futhark store this in memory?

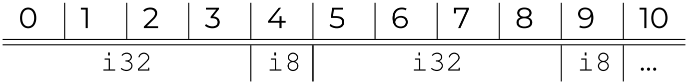


<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td></tr>
<tr><td colspan="4">i32</td><td>i8</td><td colspan="4">i32</td><td>i8</td><td>...</td></tr>
</table>

Problem: Unaligned access.

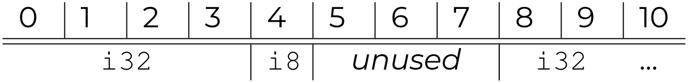
Arrays of tuples

Consider arrays of type `[](i32, i8)`. Since an `i32` is four bytes and a `i8` is one byte, how should Futhark store this in memory?



0	1	2	3	4	5	6	7	8	9	10
`i32`				`i8`	`i32`				`i8`	...

Problem: Unaligned access.



0	1	2	3	4	5	6	7	8	9	10
`i32`				`i8`	*unused*			`i32`		...

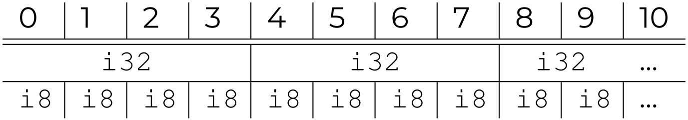
Problem: Waste of memory.

And both lead to uncoalesced access when the tuples are large.

Tuples of arrays

Representation

Array type `[n](a, b, c...)` is represented in memory as `([n]a, [n]b, [n]c...)`, i.e. as *multiple arrays*, each containing only primitive values.



0	1	2	3	4	5	6	7	8	9	10
`i32`				`i32`				`i32`		...
`i8`	`i8`	`i8`	`i8`	`i8`	`i8`	`i8`	`i8`	`i8`	`i8`	...

- Common (and crucial) transformation.
- Called “struct of arrays” in legacy languages.
- Automatically done by the Futhark compiler.
- Only affects internal language.

Higher-order functions are problematic

Higher-order functions are problematic

- Normally implemented via function pointers.
- GPUs do not efficiently support function pointers.

Fortunately, the 70s were full of people who did not like function pointers either.

(Futhark work by Anders Kiel Hovgaard)

Defunctionalisation (Reynolds, 1972)

John Reynolds: "Definitional interpreters for higher-order programming languages"

- Replace lambdas by tagged data value that captures free variables:

$$\lambda x.x + y \Longrightarrow \mathrm{Lam}l\ y$$

Defunctionalisation (Reynolds, 1972)

John Reynolds: "Definitional interpreters for higher-order programming languages"

- Replace lambdas by tagged data value that captures free variables:

$$\lambda x.x + y \Longrightarrow \mathrm{Lam}I\ y$$

- Replace application by case dispatching over these functions:

$$\begin{aligned} f\ a \quad \Longrightarrow \quad \textbf{case}\ f\ \textbf{of}\ & \mathrm{Lam1} \ldots \rightarrow \ldots \\ & \mathrm{Lam2} \ldots \rightarrow \ldots \\ & \mathrm{Lam}I\ y \rightarrow a + y \\ & \ldots \end{aligned}$$

- **Branch divergence on GPUs.**
- **Arrays of these things are likely inefficient.**

Ensuring branch-free defunctionalisation

Conditionals may not produce functions:

```
let f = if b1 then \x -> foo
        else if b N then \x -> bar
        else ... \x -> baz
in... f y
```

- Which function `f` is applied?
- To defunctionalise without introducing branching, we must restrict conditionals from returning functions.
- We require that branches have **order zero** type.

More restrictions

Arrays may not contain functions

```
let fs = [\y -> y+a, \z -> z*b, ...]
in... fs[n] 5
```

- Which function `fs[n]` is applied?

- And a few similar restrictions for other language constructs…
- **Restricting the language enables better code generation.**
- Important: the restrictions are *easy to understand*, checked in the type checker, and are often not a hindrance in practice

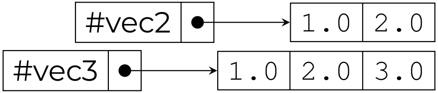
Sum types (work by Robert Schenk)

Sum types (work by Robert Schenk)

The usual representation is tag plus pointer

```
-- Constructor names are #-prefixed in Futhark
type vec = #vec2 {x: f32, y: f32}
         | #vec3 {x: f32, y: f32, z: f32}
```

```
#vec2 {x=1, y=2}:

#vec3 {x=1, y=2, z=3}:
```



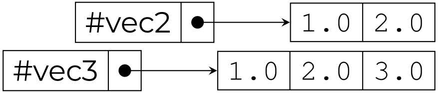
Composes well, and never uses more space than necessary.

Sum types (work by Robert Schenk)

The usual representation is tag plus pointer

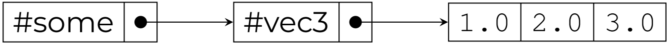
```
-- Constructor names are #-prefixed in Futhark
type vec = #vec2 {x: f32, y: f32}
         | #vec3 {x: f32, y: f32, z: f32}
```

```
#vec2 {x=1, y=2}:

#vec3 {x=1, y=2, z=3}:
```



Composes well, and never uses more space than necessary.

```
type opt 'a = #some a | #none

#some (#vec2 {x=1, y=2, z=3}):
```



Problems on GPU

Irregular representation requires unpredictable allocations

```
if x >= 0 then #some (sqrt x)
           else #none
```

Where do we get the memory for the `#some` payload?

Problems on GPU

Irregular representation requires unpredictable allocations

```
if x >= 0 then #some (sqrt x)
          else #none
```

Where do we get the memory for the `#some` payload?

Memory access becomes uncoalesced

```
[#some 1, #none, #some 3, ...]
```

When `map`-ing, no guarantee that the payloads are adjacent in memory.

Falling back to a solved problem

We translate sum types to *tuples*.

```
type vec2 = {x: f32, y: f32}
type vec3 = {x: f32, y: f32, z: f32}
type vec = #vec2 vec2 | #vec3 vec3
```

becomes

```
type vec = (i8, vec2, vec3)
```

with the `i8` encoding the constructor.

```
#vec2 {x=1,y=2}     ⇒ (0, {x=1,y=2}, {x=0,y=0,z=0})
#vec3 {x=1,y=2,z=3} ⇒ (1, {x=0,y=0}, {x=1,y=2,z=3})
```

Insert **dummy values** for unused constructor payloads.

Union payloads

Rust implements sum types by making their payload the size of the *maximum* of the constructor payloads.

tag	payload bytes

```
#some (#vec3 {x=1.0, y=2.0, z=3.0})
```

#some	#vec3	1.0	2.0	3.0

```
#some (#vec2 {x=1.0, y=2.0})
```

#some	#vec2	1.0	2.0	-

```
#none
```

#none

Union payloads

Rust implements sum types by making their payload the size of the *maximum* of the constructor payloads.

tag	payload bytes

```
#some (#vec3 {x=1.0, y=2.0, z=3.0})
```

#some	#vec3	1.0	2.0	3.0

```
#some (#vec2 {x=1.0, y=2.0})
```

#some	#vec2	1.0	2.0	-

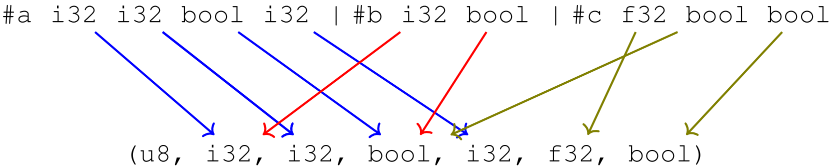
```
#none
```

#none

Unfortunately doesn’t work with the tuple-of-arrays transformation.

Deduplication

Deduplication exploits redundancies across constructors



`#a i32 i32 bool i32 | #b i32 bool | #c f32 bool bool`

`(u8, i32, i32, bool, i32, f32, bool)`

Impact of deduplication

Deduplication gives 2 × **speedup on a ray tracer**

```
type vec3 = {x: f32, y: f32, z: f32}

type material = #lambertian {albedo: vec3}
              | #metal {albedo: vec3, fuzz: f32}
              | #dielectric {ref_idx: f32}
```

`https://github.com/athas/raytracinginoneweekendinfuthark`

Matrix multiplication

Matrix multiplication

```
def dotprod x y = f32.sum (map2 (*) x y)

def matmul A B =
  map (\A_row -> map (\B_col -> dotprod A_row B_col)
                     (transpose B))
      A
```

Matrix multiplication

```
def dotprod x y = f32.sum (map2 (*) x y)

def matmul A B =
  map (\A_row -> map (\B_col -> dotprod A_row B_col)
                     (transpose B))
      A
```

Multiplying 4096 × 1024 and 1024 × 4096 matrices on A100 GPU

Futhark: $3880\mu s$

Matrix multiplication

```
def dotprod x y = f32.sum (map2 (*) x y)

def matmul A B =
  map (\A_row -> map (\B_col -> dotprod A_row B_col)
                     (transpose B))
      A
```

Multiplying 4096 × 1024 and 1024 × 4096 matrices on A100 GPU

Futhark: 3880μs

cuBLAS: 1899μs (2 × faster than Futhark)

Application performance

Benchmarks based on the Monte Carlo neutron transport algorithm.

XSBench Futhark:
Original:

RSBench Futhark:
Original:

Ported from hand-written CUDA to Futhark.

Application performance

Benchmarks based on the Monte Carlo neutron transport algorithm.

XSBench
Futhark: 142*ms*
Original: 142*ms*

RSBench
Futhark: 1342*ms*
Original: 1108*ms* (1.21 × faster than Futhark)

Ported from hand-written CUDA to Futhark.

Conclusions

- Functional programming *is* good for parallelism.
  - But many of its classical features are not.
- Locality and regularity are central to truly high performance.
  - And functional programming makes it easy to build irregular things.
- Optimisations and transformations are known that can help.
  - But they make tradeoffs that are not right for every situation.
- **Things are much easier when you restrict the input language.**

Go try Futhark! `https://futhark-lang.org`