

# Regular flattening

Troels Henriksen (athas@sigkill.dk)

DIKU  
University of Copenhagen

23rd of November, 2020

# Agenda

Representation and Fusion

Handling nested parallelism

Basic flattening rules

Incremental flattening

Multi-level parallelism

Final words as time permits

## Representation and Fusion

Handling nested parallelism

Basic flattening rules

Incremental flattening

Multi-level parallelism

Final words as time permits

## Representing arrays of tuples

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

## Representing arrays of tuples

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

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

**Problem?**

## Representing arrays of tuples

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

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

**Problem?** Unaligned accesses.

# Representing arrays of tuples

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

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

**Problem?** Unaligned accesses.

0	1	2	3	4	5	6	7	8	9	10
i32				i8	<i>unused</i>				i32	...

**Problem?**

# Representing arrays of tuples

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

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

**Problem?** Unaligned accesses.

0	1	2	3	4	5	6	7	8	9	10
i32				i8	<i>unused</i>				i32	...

**Problem?** Waste of memory.



# Tuples of arrays

## Representation

An array `[](t1, t2, t3...)` is represented in memory as `([]t1, []t2, []t3...)`, 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) optimisation.
- Called “struct of arrays” in legacy languages.
- Automatically done by the Futhark compiler.
- Only affects internal language.
- ...and also assumed for the rest of today’s presentation.

## "Unzipped" SOACs

Instead of

```
let tmp = map (\(x,y) -> (x-1, y+1))  
            (zip xs ys)  
let (xs, ys) = unzip xs_ys'
```

we write

```
let (xs, ys) = map (\x y -> (x-1, y+1)) xs ys
```

- In the compiler IR, **All SOACs accept multiple array inputs and produce unzipped results.**
- Arrays of tuples (or records, or sums) do not exist in the core language.
- **Isomorphic to source language**, but this form is much easier to work with in a compiler.

## Loop fusion

```
let increment [n][m] (as: [n][m]i32) : [n]i32 =  
  map (\r -> map (+2) r) a  
let sum [n] (a: [n]i32) : i32 =  
  reduce (+) 0 a  
let sumrows [n][m] (as: [n][m]i32) : [n]i32 =  
  map sum as
```

Let's say we wish to first call increment, then sumrows:

sumrows (increment a)

**Naively** Run increment, then call sumrows.

**Problem** Manifests intermediate matrix in memory.

**Solution** *Loop fusion*, which combines loops to avoid intermediate results.

## An example of a fusion rule

The expression

**map**  $f$  (**map**  $g$   $a$ )

is *always* equivalent to

**map**  $(f \circ g)$   $a$

- This is an extremely powerful property that is only true in the absence of side effects.
- Fusion is *the* core optimisation that permits the efficient decomposition of a data-parallel program.
- A full fusion engine has much more awkward-looking rules (zip/unzip causes lots of bookkeeping), but safety is guaranteed.

## A fusion example

<code>sumrows (increment a) =</code>	(Initial expression)
<code>map sum (increment a) =</code>	(Inline sumrows)
<code>map sum (map (<math>\lambda r \rightarrow</math> map (+2) r) a) =</code>	(Inline increment)
<code>map (sum <math>\circ</math> (<math>\lambda r \rightarrow</math> map (+2) r) a) =</code>	(Apply <b>map-map</b> fusion)
<code>map (<math>\lambda r \rightarrow</math> sum (map (+2) r) a) =</code>	(Apply composition)

- We have avoided the temporary matrix, but the composition of `sum` and the **map** also holds an opportunity for fusion – specifically, **reduce-map** fusion.
- Will not cover in detail, but a **reduce** can efficiently apply a function to each input element before engaging in the actual reduction operation.
- Important to remember: a **map** going into a **reduce** is an efficient pattern.

# A shorthand notation for sequences

$$\bar{z}^{(n)} = z_0, \dots, z_{(n-1)}$$

- The  $n$  may be omitted.
- A separator may be implied by context.

$$f \bar{v}^{(n)} \equiv f v_1 \dots v_n$$

or a tuple

$$(\bar{v}^{(n)}) \equiv (v_1, \dots, v_n)$$

or a function type

$$\bar{\tau}^{(n)} \rightarrow \tau_{n+1} \equiv \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau_{n+1}.$$

When not all terms under the bar are variant, subscript variant terms with  $i$ .

$$(\overline{[d]v_i}^{(n)}) = ([d]v_1, \dots, [d]v_n)$$

and

$$(\overline{[d_i]v_i}^{(n)}) = ([d_1]v_1, \dots, [d_n]v_n)$$

# Fused constructs

## Convenient shorthands

$$\mathbf{redomap} \odot f \bar{d} \overline{xs} \equiv$$

$$\mathbf{reduce} \odot \bar{d} (\mathbf{map} f \overline{xs})$$

$$\mathbf{scanomap} \odot f \bar{d} \overline{xs} \equiv$$

$$\mathbf{scan} \odot \bar{d} (\mathbf{map} f \overline{xs})$$

- Emphasises that **reduce**/**scan-map** compositions can be considered as a single construct.
- We will see several examples where this is useful.

# Fused constructs

## Convenient shorthands

$$\mathbf{redomap} \odot f \bar{d} \overline{xs} \equiv$$

$$\mathbf{reduce} \odot \bar{d} (\mathbf{map} f \overline{xs})$$

$$\mathbf{scanomap} \odot f \bar{d} \overline{xs} \equiv$$

$$\mathbf{scan} \odot \bar{d} (\mathbf{map} f \overline{xs})$$

- Emphasises that **reduce/scan-map** compositions can be considered as a single construct.
- We will see several examples where this is useful.

### Note:

$$\mathbf{reduce} \odot \bar{d} \overline{xs} \equiv$$

$$\mathbf{reduce} \odot \bar{d} (\mathbf{map} \mathbf{id} \overline{xs}) \equiv$$

$$\mathbf{redomap} \odot f \bar{d} \overline{xs}$$

$$\mathbf{scan} \odot \bar{d} \overline{xs} \equiv$$

$$\mathbf{scan} \odot \bar{d} (\mathbf{map} \mathbf{id} \overline{xs}) \equiv$$

$$\mathbf{scanomap} \odot f \bar{d} \overline{xs}$$



Representation and Fusion

Handling nested parallelism

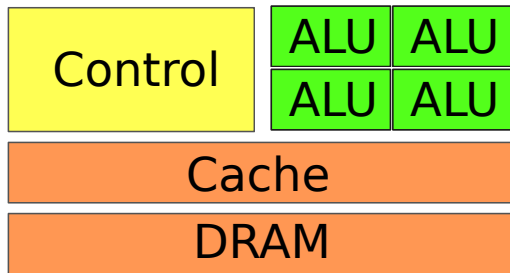
Basic flattening rules

Incremental flattening

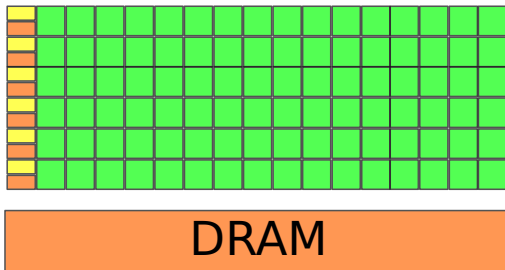
Multi-level parallelism

Final words as time permits

# GPUs vs CPUs



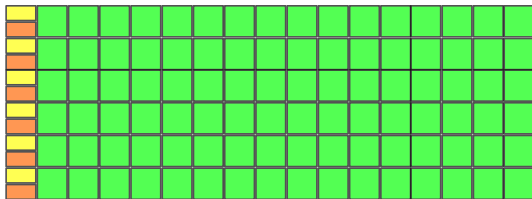
CPU



GPU

- 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*.

# The SIMT Programming Model



- GPUs are programmed using the SIMT model (*Single Instruction Multiple Thread*).
- Similar to SIMD (*Single Instruction Multiple Data*), but while SIMD has explicit vectors, we provide *sequential scalar per-thread* code in SIMT.

Each thread has its own registers, but they all execute the same instructions at the same time (i.e. they share their instruction pointer).

## SIMT example

For example, to increment every element in an array *a*, we might use this code:

```
increment(a) {  
    tid = get_thread_id();  
    x = a[tid];  
    a[tid] = x + 1;  
}
```

- If *a* has *n* elements, we launch *n* threads, with `get_thread_id()` returning *i* for thread *i*.
- This is *data-parallel programming*: applying the same operation to different data.
- When we launch a GPU program (*kernel*), we say how many threads should be launched, *all running the same code*.

# Branching

If all threads share an instruction pointer, what about branches?

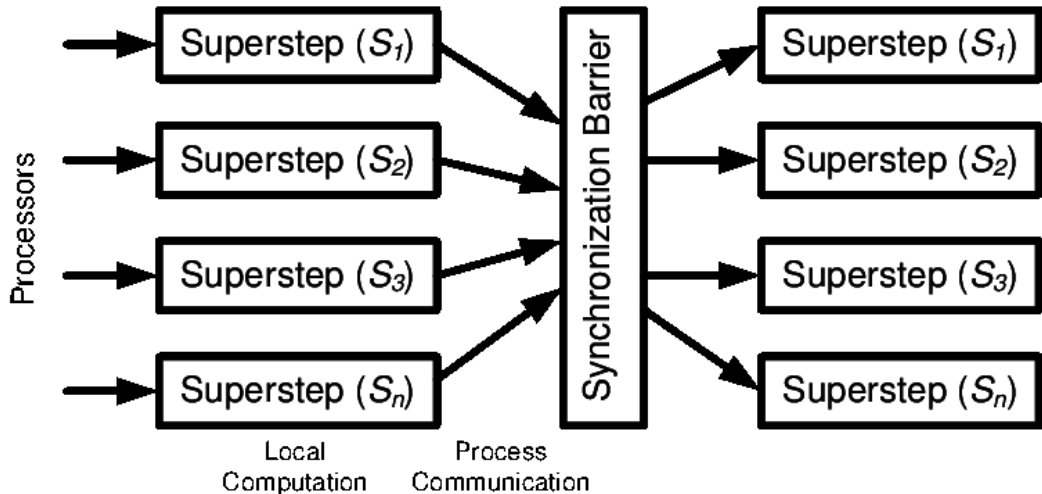
```
mapabs(a) {  
    tid = get_thread_id();  
    x = a[tid];  
    if (x < 0) {  
        a[tid] = -x;  
    }  
}
```

## Masked Execution

Both branches are executed in all threads, but in those threads where the condition is false, a mask bit is set to treat the instructions inside the branch as no-ops.

## Do GPUs exist in theory as well?

GPU programming is a close fit to *bulk synchronous parallelism*:



- Supersteps are *threads*, which cannot talk to each other.

## A SOAC-kernel correspondence

The compiler *knows*<sup>2</sup> that certain nests of perfect **maps** correspond to certain GPU basic blocks.

- **maps** containing scalar code is a kernel with one thread per iteration of the **maps**.
- **maps** containing a single **reduce** is a *segmented reduction*.
- **maps** containing a single **scan** is a *segmented scan*.
- **maps** containing a single **scatter** is a *segmented scatter*.
- ...see the pattern?

**Crucial:** the **maps** must be *perfectly nested*.

```
map (\xs y -> map (\x -> x + y) xs) xss ys
```

Suppose `xss` is of shape `[n][m]`, then this can compile to a kernel with  $n \times m$  threads, each doing a single  $x + y$  operation.

---

<sup>2</sup>Because it has taken PMPH.

# Handling nested parallelism

## Problem

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



# Handling nested parallelism

## Problem

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

## Solution

Have the compiler rewrite program to perfectly nested **maps** containing sequential code, or known parallel patterns such as segmented reduction.

# Handling nested parallelism

## Problem

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

## Solution

Have the compiler rewrite program to perfectly nested **maps** containing sequential code, or known parallel patterns such as segmented reduction.

```
map (\xs -> let y = reduce (+) 0 xs
      in map (\x -> x + y) xs)
  xss
```

⇓

```
let ys = map (\xs -> reduce (+) 0 xs) xss
in map (\xs y -> map (\x -> x + y) xs) xss ys
```

# Flattening via loop fission

The classic map fusion rule:

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

---

<sup>3</sup>*Futhark: Purely Functional GPU-Programming with Nested Parallelism and In-Place Array Updates*, PLDI 2017

# 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<sup>3</sup>, or just wait until later in the lecture), are applied by the compiler to extract perfect map nests.

---

<sup>3</sup>Futhark: Purely Functional GPU-Programming with Nested Parallelism and In-Place Array Updates, PLDI 2017

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

```
let (asss , bss) =  
  map (\(ps: [m]i32) ->  
    let ass = map (\(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 (\as 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 (\(ps: [m]i32) ->  
    let ass = map (\(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 (\ps ass ->  
    let bs = loop ws=ps for i < n do  
      map (\as 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 (\(ps: [m]i32) ->  
    let ass = map (\(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 (\ps ass ->  
    let bs = loop ws=ps for i < n do  
      map (\as 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 (\(ps: [m]i32) ->  
    let ass = map (\(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 (\ass ws ->  
      let ws' = map (\as 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 (\(ps: [m]i32) ->  
    let ass = map (\(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 (\ass ws ->  
      let ws' = map (\as 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 (\(ps: [m]i32) ->  
    let ass = map (\(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 (\ass ws ->  
      let ws' = map (\as 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 (\(ps: [m]i32) ->  
    let ass = map (\(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 (\ass ws ->  
      let ws' = map (\as 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 (\(ps: [m]i32) ->  
    let ass = map (\(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 (\ass ->  
        map (\as ->  
          reduce (+) 0 as) ass)  
        asss  
    in map (\ws ds ->  
      let ws' =  
        map (\w d -> let e = d + w  
          in 2 * e) ws ds  
      in ws') asss dss
```

## (f) Distributing inner map

```
let asss =  
  map (\(ps: [m]i32) ->  
    let ass = map (\(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 (\(ps: [m]i32) ->  
    let rss = map (\(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 (\(ps: [m]i32) (rs: [m]i32) ->  
    map (\(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 (\(ps: [m]i32) ->  
    let rss = map (\(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 (\(ps: [m]i32) ->  
    let rss = map (\(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 (\ps -> map (...) ps) pss
let asss: [m][m][m]i32 =
  map (\ps rs -> map (\r -> map (...) ps) rs) pss rss
let bss: [m][m]i32 =
  loop wss=pss for i < n do
    let dss: [m][m]i32 = map (\ass -> map (reduce ...) ass)
                          asss
    in map (\ws 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.

Representation and Fusion

Handling nested parallelism

**Basic flattening rules**

Incremental flattening

Multi-level parallelism

Final words as time permits

# Notation for flat parallelism

## Instead of writing

```
map (\ps rs ->  
    map (\r ->  
        map (\p -> e)  
            ps)  
        rs)  
    pss rss
```

## We write

**segmap** ( $\langle ps, rs \in pss, rss \rangle$ ,  $\langle r \in rs \rangle$ ,  $\langle p \in ps \rangle$ )  
 $e$

## Segmented flat parallel constructs

$$\Sigma = \Sigma', \langle \bar{x} \in \bar{y} \rangle$$

$$\begin{aligned} \mathbf{segmap} \Sigma e \equiv & \mathbf{map} (\lambda \bar{x}_p \rightarrow \\ & \mathbf{map} (\lambda \overline{x_{p-1}} \rightarrow \dots \\ & \mathbf{map} (\lambda \overline{x_1} \rightarrow e) \overline{y_1}) \\ & \overline{y_{p-1}}) \\ & \overline{y_p} \end{aligned}$$

- Conceptually a stack of **map**s with some parallel construct (here another **map**) inside.
- *These* are what triggers GPU code generation.
- Any SOACs left in  $e$  will be executed sequentially.

## Similarly for reductions and scans

$$\begin{aligned} \mathbf{segred} \Sigma \odot \bar{d} e \equiv & \mathbf{map} (\lambda \bar{x}_p \rightarrow \\ & \mathbf{map} (\lambda \overline{x_{p-1}} \rightarrow \dots \\ & \mathbf{redomap} \odot (\lambda \bar{x}_1 \rightarrow e) \bar{d} \bar{y}_1) \\ & \overline{y_{p-1}}) \\ & \bar{y}_p \end{aligned}$$

$$\begin{aligned} \mathbf{segscan} \Sigma \odot \bar{d} e \equiv & \mathbf{map} (\lambda \bar{x}_p \rightarrow \\ & \mathbf{map} (\lambda \overline{x_{p-1}} \rightarrow \dots \\ & \mathbf{scanomap} \odot (\lambda \bar{x}_1 \rightarrow e) \bar{d} \bar{y}_1) \\ & \overline{y_{p-1}}) \\ & \bar{y}_p \end{aligned}$$

Let us look at how one can rewrite SOAC nests to these segmented operations.

## Example of rewrite rules

Rules describe how valid *judgments* can be formed.

### Example with partial evaluation

$\mathcal{V} \vdash e_1 \Rightarrow e_2$  where  $\mathcal{V}$  is a mapping from variable names  $v$  to values.

$$\frac{}{\mathcal{V} \vdash e_1 \Rightarrow e_2}$$

$$\frac{}{\mathcal{V} \vdash v \Rightarrow \mathcal{V}(v)}$$

$$\frac{\mathcal{V} \vdash e_1 \Rightarrow \text{true}}{\mathcal{V} \vdash \mathbf{if} \ e_1 \ \mathbf{then} \ e_2 \ \mathbf{else} \ e_3 \Rightarrow e_2}$$

## Example of rewrite rules

Rules describe how valid *judgments* can be formed.

### Example with partial evaluation

$\boxed{\mathcal{V} \vdash e_1 \Rightarrow e_2}$  where  $\mathcal{V}$  is a mapping from variable names  $v$  to values.

$$\begin{array}{c} \frac{}{\mathcal{V} \vdash e_1 \Rightarrow e_2} \qquad \frac{}{\mathcal{V} \vdash v \Rightarrow \mathcal{V}(v)} \qquad \frac{\mathcal{V} \vdash e_1 \Rightarrow \text{true}}{\mathcal{V} \vdash \mathbf{if} \ e_1 \ \mathbf{then} \ e_2 \ \mathbf{else} \ e_3 \Rightarrow e_2} \\[10pt] \frac{\mathcal{V}, x \mapsto e_1 \vdash e_2 \Rightarrow e'_2}{\mathcal{V} \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 \Rightarrow \mathbf{let} \ x = e_1 \ \mathbf{in} \ e'_2} \qquad \frac{x \notin FV(e_2)}{\mathcal{V} \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 \Rightarrow e_2} \end{array}$$

- Rewrite rules can be ambiguous (several may apply).
- Need a decision procedure in order to have an *algorithm*.

# Flattening rules

$\boxed{\Sigma \vdash e \Rightarrow e'}$  In a map-nest context  $\Sigma$ , the source expression  $e$  can be translated into the target expression  $e'$ .

$$\frac{\begin{array}{c} e \text{ has inner SOACs} \\ \Sigma, \langle \bar{x} \in \overline{xs} \rangle \vdash e \Rightarrow e_{\text{flat}} \end{array}}{\Sigma \vdash \mathbf{map} (\lambda \bar{x} \rightarrow e) \overline{xs} \Rightarrow e_{\text{flat}}}$$

$$\frac{\text{no other rule applies}}{\bullet \vdash e \Rightarrow e}$$

$$\frac{\Sigma \neq \bullet}{\Sigma \vdash e \Rightarrow \mathbf{segmap} \Sigma e}$$

---


$$\Sigma \vdash \mathbf{redomap} \odot (\lambda \bar{x} \rightarrow e) \bar{d} \overline{xs} \Rightarrow \mathbf{segred} (\Sigma, \bar{x} \in \overline{xs}) \odot \bar{d} e$$



## Rule for map distribution

$$\frac{\begin{array}{l} \text{size of each array in } \overline{a_0} \text{ invariant to } \Sigma \\ \Sigma = \langle \overline{x_p} \in \overline{y_p} \rangle, \dots, \langle \overline{x_1} \in \overline{y_1} \rangle \quad \Sigma \vdash e_1 \Rightarrow e'_1 \\ \overline{a_p}, \dots, \overline{a_1} \text{ fresh names} \quad \Sigma' \vdash e_2 \Rightarrow e'_2 \\ \Sigma' = \langle \overline{x_p} \overline{a_{p-1}} \in \overline{y_p} \overline{a_p} \rangle, \dots, \langle \overline{x_1} \overline{a_0} \in \overline{y_1} \overline{a_1} \rangle \end{array}}{\Sigma \vdash \mathbf{let} \overline{a_0} = e_1 \mathbf{in} e_2 \Rightarrow \mathbf{let} \overline{a_p} = e'_1 \mathbf{in} e'_2}$$

# Rule for map distribution

$$\begin{array}{c}
\text{size of each array in } \overline{a_0} \text{ invariant to } \Sigma \\
\Sigma = \langle \overline{x_p} \in \overline{y_p} \rangle, \dots, \langle \overline{x_1} \in \overline{y_1} \rangle \quad \Sigma \vdash e_1 \Rightarrow e'_1 \\
\overline{a_p}, \dots, \overline{a_1} \text{ fresh names} \quad \Sigma' \vdash e_2 \Rightarrow e'_2 \\
\Sigma' = \langle \overline{x_p} \overline{a_{p-1}} \in \overline{y_p} \overline{a_p} \rangle, \dots, \langle \overline{x_1} \overline{a_0} \in \overline{y_1} \overline{a_1} \rangle \\
\hline
\Sigma \vdash \mathbf{let} \ \overline{a_0} = e_1 \ \mathbf{in} \ e_2 \Rightarrow \mathbf{let} \ \overline{a_p} = e'_1 \ \mathbf{in} \ e'_2
\end{array}$$

## Example for

```
map (\xs -> let y = redomap (+) (\x -> x) 0 xs
         in map (\x -> x + y) xs)
  xss
```

## Suppose already inside the outer map

$$\begin{array}{l}
\Sigma = \langle xs \in xss \rangle \quad \Sigma' = \langle xs, y \in xss, ys \rangle \\
\Sigma \vdash \text{redomap } (+) \ (\lambda x \rightarrow x) \ 0 \ xs \Rightarrow \mathbf{segred} \ (\Sigma, \langle x \in xs \rangle) \ (+) \ 0 \ x \\
\Sigma' \vdash \text{map } (\lambda x \rightarrow x + y) \ xs \Rightarrow \mathbf{segmap} \ (\Sigma', \langle x \in xs \rangle) \ x + y \\
\Sigma \vdash \dots \Rightarrow \mathbf{let} \ ys = \mathbf{segred} \ (\langle xs \in xss \rangle, \langle x \in xs \rangle) \ (+) \ 0 \ x \\
\mathbf{in} \ \mathbf{segmap} \ \langle xs, y \in xss, ys \rangle \ x + y
\end{array}$$

## Handling transposition

**rearrange**  $(d_1, \dots, d_n)$   $x$  is a generalization of **transpose** in that it rearranges the dimensions of  $d$ -dimensional array based on a permutation defined by the integer sequence  $d_1, \dots, d_n$ . E.g:

$$\mathbf{transpose} \equiv \mathbf{rearrange} (1, 0)$$

# Handling transposition

**rearrange**  $(d_1, \dots, d_n) x$  is a generalization of **transpose** in that it rearranges the dimensions of  $d$ -dimensional array based on a permutation defined by the integer sequence  $d_1, \dots, d_n$ . E.g:

$$\mathbf{transpose} \equiv \mathbf{rearrange} (1, 0)$$

## Flattening rule

$$\frac{\Sigma \vdash \mathbf{rearrange} (0, 1 + k_1, \dots, 1 + k_n) y \Rightarrow e}{\Sigma, \langle x \in y \rangle \vdash \mathbf{rearrange} (k_1, \dots, k_n) x \Rightarrow e}$$

# Handling transposition

**rearrange**  $(d_1, \dots, d_n)$   $x$  is a generalization of **transpose** in that it rearranges the dimensions of  $d$ -dimensional array based on a permutation defined by the integer sequence  $d_1, \dots, d_n$ . E.g:

$$\mathbf{transpose} \equiv \mathbf{rearrange} (1, 0)$$

## Flattening rule

$$\frac{\Sigma \vdash \mathbf{rearrange} (0, 1 + k_1, \dots, 1 + k_n) y \Rightarrow e}{\Sigma, \langle x \in y \rangle \vdash \mathbf{rearrange} (k_1, \dots, k_n) x \Rightarrow e}$$

## Example

- $\vdash \mathbf{map} (\lambda x \rightarrow \mathbf{rearrange} (1, 0) x) xs \Rightarrow \mathbf{rearrange} (0, 2, 1) xs$

# map-loop-interchange

$f$  contains exploitable (regular) parallelism

$\Sigma' = \Sigma, \langle \bar{x} \bar{y} \in \bar{x} \bar{s} \bar{y} \bar{s} \rangle$

$\bar{z} s', \bar{y} s'$  fresh names

$m = \text{outer size of each of } \bar{x} \bar{s} \text{ and } \bar{y} \bar{s}$

$\bar{z} \bar{r} \equiv \text{replicate } m \ z_i$

$\{n, \bar{q}, \bar{z}\} \cap \{\bar{x}, \bar{y}\} = \emptyset$

$\Sigma \vdash_l \text{loop } \bar{z} s' \ \bar{y} s' = \bar{z} \bar{r} \ \bar{y} \bar{s} \text{ for } i < n \text{ do map } (f \ i \ \bar{q}) \ \bar{x} \bar{s} \ \bar{y} \bar{s} \ \bar{y} s' \ \bar{z} s' \Rightarrow e$

---

$\Sigma' \vdash_l \text{loop } \bar{z} \bar{r} \ \bar{y} \bar{r} = \bar{z} \bar{y} \text{ for } i < n \text{ do } f \ i \ \bar{q} \ \bar{x} \bar{y} \ \bar{y} \bar{r} \ \bar{z} \bar{r} \Rightarrow e$

## Informal example

```
map (\xs -> loop (xs', j) = (xs, 0) for i < n do
    (map (+j) xs', j + i))
  xss
```

*Becomes after interchange*

```
loop (xss', js) = (xss, replicate m 0) for i < n do
  map (\xs' j -> (map (+j) xs', j + i))
    xss' j
```

## Validity of interchange

The simple intuition is that

`map (\x -> loop x' = x for i < n do (f x')) xs`

is equivalent to

`loop xs' = xs for i < n do (map f xs')`

because they both produce

$$[f^n xs[0], \dots, f^n xs[m-1]]$$

Representation and Fusion

Handling nested parallelism

Basic flattening rules

**Incremental flattening**

Multi-level parallelism

Final words as time permits



## Consider Matrix Multiplication

```
for i < n:  
    for j < m:  
        acc = 0  
        for l < p:  
            acc += xss[i,l] * yss[l,j]  
        res[i,j] = acc
```

## Turning it Functional

```
map (\xs ->  
    map (\ys ->  
        let zs = map (*) xs ys  
        in reduce (+) 0 zs)  
    (transpose yss))  
xss
```

## Using **redomap** notation

```
map (\xs ->  
    map (\ys ->  
        redomap (+) (*) 0 xs ys)  
        (transpose yss))  
xss
```

$$\mathbf{redomap} \odot f \circ x \equiv \mathbf{reduce} \odot 0 \circ (\mathbf{map} f x)$$

Emphasises that a **map-reduce** composition can be turned into a fused tight sequential loop, or into a parallel reduction.

**So how should we parallelise this on GPU?**

## So how should we parallelise this on GPU?

*Full flattening*

```
map (\ xs ->  
  map (\ ys ->  
    redomap (+) (*) 0  
            xs ys)  
    (transpose yss))  
xss
```

- **All parallelism exploited**
- Some communication overhead
- *Best if outer **maps** don't saturate GPU*

# So how should we parallelise this on GPU?

## Full flattening

```
map (\ xs ->  
    map (\ ys ->  
        redomap (+) (*) 0  
              xs ys)  
      (transpose yss))  
xss
```

- All parallelism exploited
- Some communication overhead
- *Best if outer **maps** don't saturate GPU*

## Moderate flattening

```
map (\ xs ->  
    map (\ ys ->  
        redomap (+) (*) 0  
              xs ys)  
      (transpose yss))  
xss
```

- Only outer parallelism
  - The **redomap** can be block tiled
  - *Best if outer **maps** saturate GPU*
- 
- There is no *one size fits all*.
  - Both situations may be encountered at program runtime.

## The essence of *incremental flattening*

**From a single source program, for each parallel construct generate multiple *semantically equivalent* parallelisations, and generate a *single program* that at runtime picks the *least parallel* that still saturates the hardware.**

- Implemented in the Futhark compiler.
- ...but technique is applicable to any (regular) nested parallelism expressed with the common Bird-Meertens-style array constructs (map, reduce, scan, etc).

# Simple Incremental Flattening

At every level of map-nesting we have two options:

1. Continue flattening inside the map, exploiting the parallelism there.
2. Sequentialise the map body; exploiting only the parallelism on top.
  - **Full flattening** in the Blelloch style will do the former, maximising utilised parallelism.
  - **Incremental flattening** generates *both* versions and uses a predicate to pick at runtime.



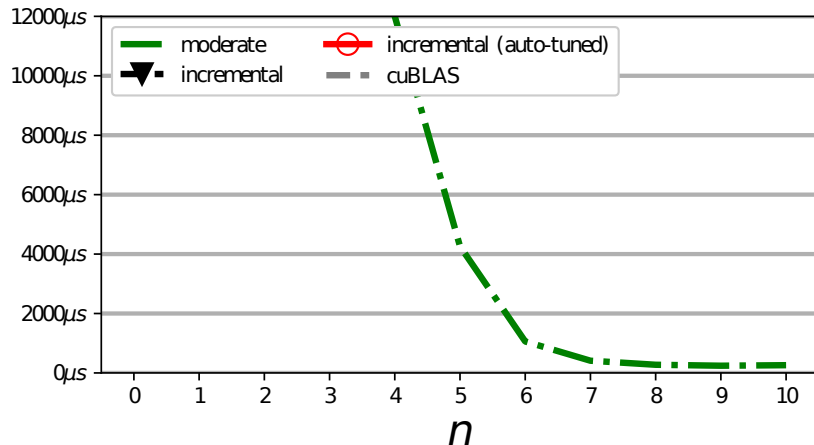
# Multi-versioned matrix multiplication

```
xss : [n][p]i32
yss : [p][m]i32.

if n * m > t0 then
  map (\xs ->
    map (\ys ->
      redomap (+) (*) 0 xs ys)
      (transpose yss))
    xss
else
  map (\xs ->
    map (\ys ->
      redomap (+) (*) 0 xs ys)
      (transpose yss))
    xss
```

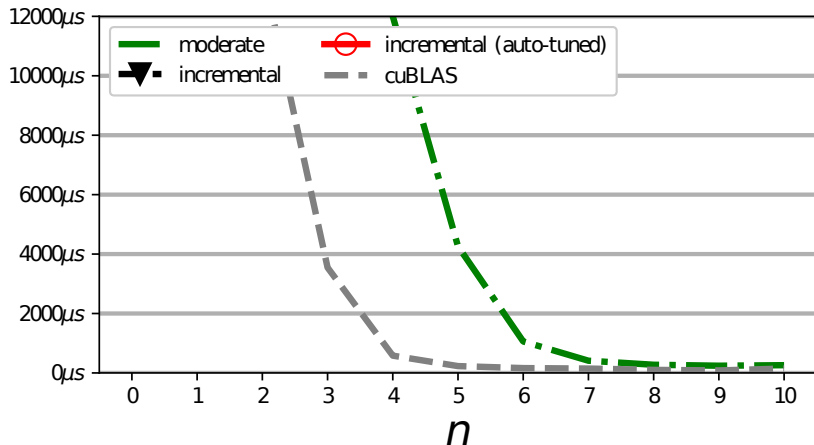
The  $t_0$  *threshold parameter* is used to select between the two versions—and should be auto-tuned on the concrete hardware.

# Matrix multiplication on NVIDIA K40



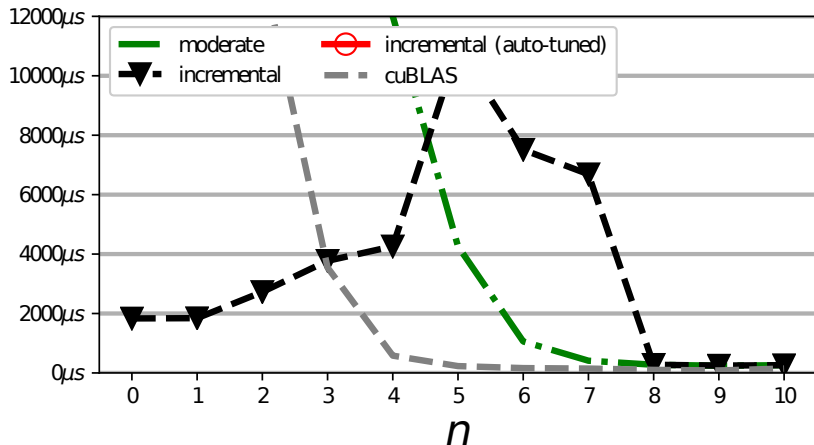
Multiplying matrices of size  $2^n \times 2^m$  and  $2^m \times 2^n$ , where  $m = 25 - 2n$ , meaning that work is constant as we vary  $n$ .

# Matrix multiplication on NVIDIA K40



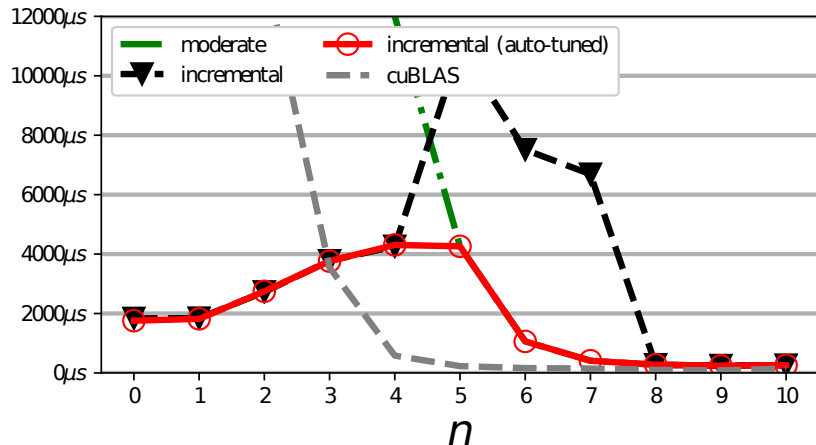
Multiplying matrices of size  $2^n \times 2^m$  and  $2^m \times 2^n$ , where  $m = 25 - 2n$ , meaning that work is constant as we vary  $n$ .

# Matrix multiplication on NVIDIA K40



Multiplying matrices of size  $2^n \times 2^m$  and  $2^m \times 2^n$ , where  $m = 25 - 2n$ , meaning that work is constant as we vary  $n$ .

# Matrix multiplication on NVIDIA K40



Multiplying matrices of size  $2^n \times 2^m$  and  $2^m \times 2^n$ , where  $m = 25 - 2n$ , meaning that work is constant as we vary  $n$ .

# Incremental flattening rule

$$\frac{\Sigma' = \Sigma, \langle \bar{x} \in \overline{xs} \rangle \quad \Sigma' \vdash e \Rightarrow e_{\text{flat}}}{\Sigma \vdash \mathbf{map} (\lambda \bar{x} \rightarrow e) \overline{xs} \Rightarrow \mathbf{if} \text{Par}(\Sigma') \geq t_{\text{top}} \mathbf{then segmap} \Sigma' e \mathbf{else} e_{\text{flat}}}$$

## Example for

`map (\xs -> redomap (+) (\x -> x) 0 xs) xss`

# Incremental flattening rule

$$\frac{\Sigma' = \Sigma, \langle \bar{x} \in \overline{xs} \rangle \quad \Sigma' \vdash e \Rightarrow e_{\text{flat}}}{\Sigma \vdash \mathbf{map} (\lambda \bar{x} \rightarrow e) \overline{xs} \Rightarrow \mathbf{if} \text{Par}(\Sigma') \geq t_{\text{top}} \mathbf{then} \mathbf{segmap} \Sigma' e \mathbf{else} e_{\text{flat}}}$$

## Example for

`map (\xs -> redomap (+) (\x -> x) 0 xs) xss`

$\Sigma = \bullet \quad \Sigma' = \langle xs \in xss \rangle$

$\Sigma' \vdash e \Rightarrow \mathbf{segred} (\langle xs \in xss \rangle, \langle x \in xs \rangle) (+) 0 x$

$\Sigma \vdash \dots \Rightarrow \mathbf{if} \text{length}(xss) \geq t_{\text{top}} \mathbf{then} \mathbf{segmap} \langle xs \in xss \rangle (\mathbf{redomap} (+) (\lambda x \rightarrow x) 0 xs) \mathbf{else} \mathbf{segred} (\langle xs \in xss \rangle, \langle x \in xs \rangle) (+) 0 x$

# Autotuning

- An incrementally flattened program may have dozens of threshold parameters,  $t_i$ , used to select versions at runtime.
- As we have seen, the default value ( $2^{16}$ ) is often not optimal.

A *configuration*  $P$  maps each  $t_i$  to an integer  $P(t_i)$ .

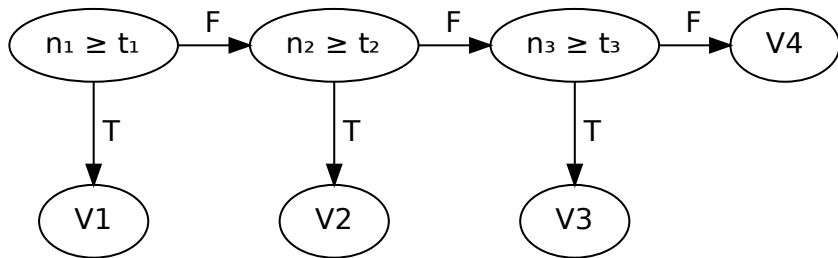
## The search problem

Find the  $P$  that minimises the cost function  $F(P)$ , where the the cost function runs the program on a set of user-provided representative datasets and sums the observed runtimes.

- Other cost functions are also possible, e.g. average runtime over datasets.
- **Note:** recompilation is not necessary.



## Briefly on our search procedure<sup>4</sup>



- Suppose we are given training data sets  $D_j, j < k$ , each of which provide a value  $v_{i,j}$  for each threshold parameter  $n_i$ .
- Starting from the deepest comparison ( $t_3$ ), for each  $D_j$  find an  $(x_j, y_j)$  that minimises runtime, take the intersection of the intervals, and use that to determine threshold value.
- Tuning time is linear in the number of comparisons.

<sup>4</sup><https://futhark-lang.org/student-projects/svend-msc-thesis.pdf>

## Using incremental flattening

Compile with a GPU backend (opencl or cuda):

```
$ futhark opencl matmul.fut
```

To autotune:

```
$ futhark autotune -v --backend=opencl matmul.fut
```

Produces `matmul.fut.tuning`, which is automatically picked up by `futhark bench` (use `--no-tuning` to stop this).

Use `futhark dev -s --extract-kernels -e matmul.fut` to see IR.

Representation and Fusion

Handling nested parallelism

Basic flattening rules

Incremental flattening

**Multi-level parallelism**

Final words as time permits

# Confession

**I lied when I claimed that GPU threads were completely isolated.**

# Confession

**I lied when I claimed that GPU threads were completely isolated.**

- Most hardware has useful (fixed) levels of parallelism.
- An ideal flattening algorithm maps levels of application parallelism (any number) to hardware parallelism (fixed number) in a way that exploits locality well.

**High-example:** a system consists of multiple *datacenters*, that each contain multiple *computers*, that each contain multiple *GPUs*, that each contain multiple *SMs* (next slide), that each run some number of threads.

# Confession

**I lied when I claimed that GPU threads were completely isolated.**

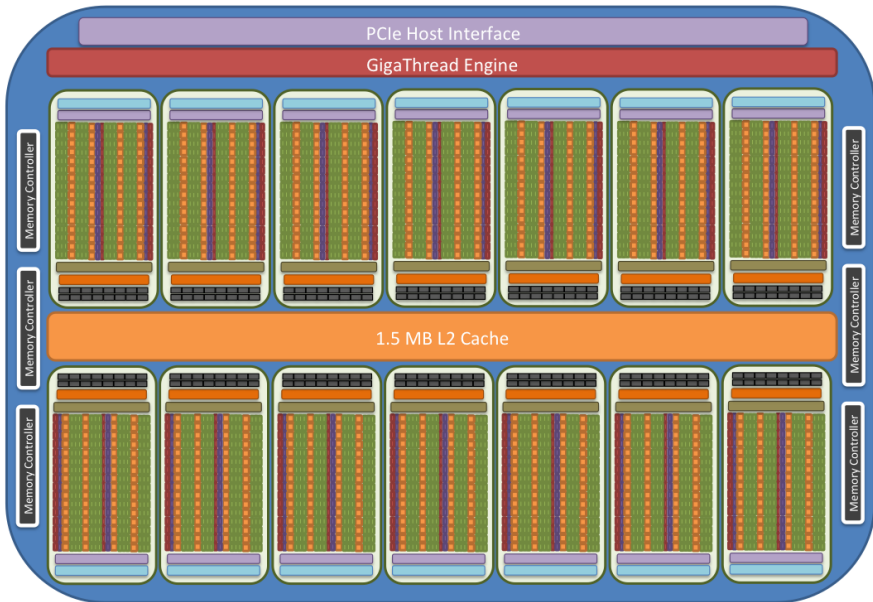
- Most hardware has useful (fixed) levels of parallelism.
- An ideal flattening algorithm maps levels of application parallelism (any number) to hardware parallelism (fixed number) in a way that exploits locality well.

**High-example:** a system consists of multiple *datacenters*, that each contain multiple *computers*, that each contain multiple *GPUs*, that each contain multiple *SMs* (next slide), that each run some number of threads.

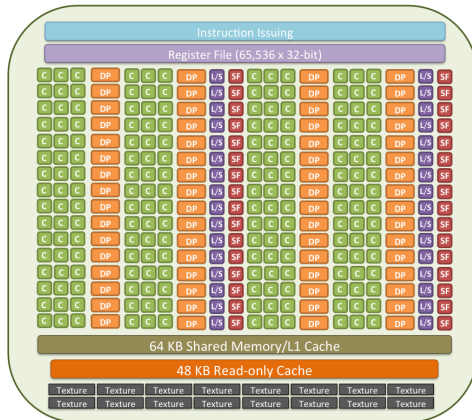
## General principle

“Tasks” at the same hardware level cannot communicate, but can “launch” tasks at a lower level.

# K20 GPU layout



# Streaming Multiprocessor (SM) layout



C

single precision/integer CUDA core

L/S

memory load/store unit

DP

double precision FP unit

SF

special function unit



# Level-aware segmented operations

$l \in \text{thread, group}$

- *Group* is the same as a CUDA *thread block*
- Each segmented operation then tagged with the level at which its *body* executes.

**segmap**<sup>*l*</sup>  $\Sigma$  *e*

**segscan**<sup>*l*</sup>  $\Sigma \odot \bar{d} \ e$

**segred**<sup>*l*</sup>  $\Sigma \odot \bar{d} \ e$

## Restrictions

Both thread and group can occur at top level, but a group construct can contain only thread constructs, and thread cannot any segmented constructs.

# Examples

**Each thread transposes part of an array**

**segmap**<sup>thread</sup>  $\langle x \in xs \rangle (\text{transpose } x)$

**Each workgroup transposes part of an array**

**segmap**<sup>group</sup>  $\langle x \in xs \rangle (\text{transpose } x)$

These are both equivalent to `map transpose xs`.

**Each workgroup sums the row of an array**

**segmap**<sup>group</sup>  $\langle xs \in xss \rangle (\text{segred}^{\text{thread}} \langle x \in xs \rangle (+) 0 x)$

Equivalent to `map (reduce (+) 0) xss`.

**Tags carry no semantic meaning; used solely for code generation.**

## Example: LocVolCalib

The following is the essential core of the LocVolCalib benchmark from the FinPar suite.

```
map (\xss ->
  map (\xs ->
    let bs = scan  $\oplus$   $d_{\oplus}$  xs
    let cs = scan  $\otimes$   $d_{\otimes}$  bs
    in scan  $\odot$   $d_{\odot}$  cs)
    xss)
  xsss
```

How can we map the application parallelism to hardware parallelism?

## Option I: sequentialise the inner scans

```
segmapthread ( $\langle xss \in xsss \rangle, \langle xs \in xss \rangle$ )  
  let bs = scan  $\oplus d_{\oplus}$  xs  
  let cs = scan  $\otimes d_{\otimes}$  bs  
  in scan  $\odot d_{\odot}$  cs
```

**scan** is relatively expensive in parallel, so this is a good option if the outer dimensions provide enough parallelism.

## Option II: flatten and parallelise inner scans

Flattening uses *loop distribution* (or *fission*) to create **map** nests:

```
map (\xss ->
  map (\xs ->
    let bs = scan  $\oplus$   $d_{\oplus}$  xs
    let cs = scan  $\otimes$   $d_{\otimes}$  bs
    in scan  $\odot$   $d_{\odot}$  cs)
    xss)
  xsss
```

## Option II: flatten and parallelise inner scans

```
let bsss =  
    segscanthread ( $\langle xss \in xsss \rangle, \langle xs \in xss \rangle, \langle x \in xs \rangle$ )  $\oplus d_{\oplus} x$   
let csss =  
    segscanthread ( $\langle bss \in bsss \rangle, \langle bs \in bss \rangle, \langle b \in bs \rangle$ )  $\oplus d_{\oplus} b$   
in  
    segscanthread ( $\langle css \in csss \rangle, \langle cs \in css \rangle, \langle c \in cs \rangle$ )  $\oplus d_{\oplus} c$ 
```

This is what full flattening will do.

## Option III: Mapping innermost parallelism to the workgroup level

```
map (\ xss ->  
  map (\ xs ->  
    let bs = scan  $\oplus$   $d_{\oplus}$  xs  
    let cs = scan  $\otimes$   $d_{\otimes}$  bs  
    in scan  $\odot$   $d_{\odot}$  cs )  
    xss )  
xsss
```

## Option III: Mapping innermost parallelism to the workgroup level

```
segmapgroup ( $\langle xss \in xsss \rangle, \langle xs \in xss \rangle$ )  
  let bs = segscanthread  $\langle x \in xs \rangle \oplus d_{\oplus} x$   
  let cs = segscanthread  $\langle b \in bs \rangle \otimes d_{\otimes} b$   
  in segscanthread  $\langle c \in cs \rangle \otimes d_{\otimes} c$ 
```

- Iterations of outer **segmaps** assigned to GPU workgroups<sup>5</sup>.
- Each **segscan**<sup>thread</sup> is executed collaboratively by a workgroup and in local memory<sup>6</sup>.
- Only works if the innermost parallelism fits in a workgroup.

---

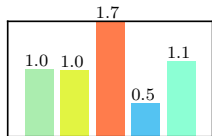
<sup>5</sup>Thread block in CUDA

<sup>6</sup>Shared memory in CUDA

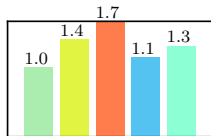


# LocVolCalib speedup (higher is better)

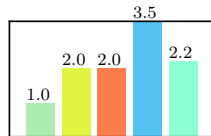
## NVIDIA K40



small (baseline: 238ms)

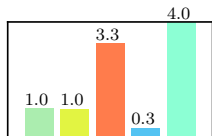


medium (baseline: 342ms)

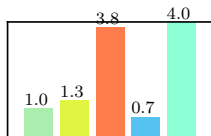


large (baseline: 5087ms)

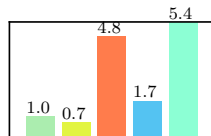
## AMD Vega 64



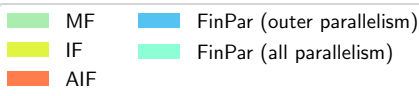
small (baseline: 110ms)



medium (baseline: 132ms)



large (baseline: 1862ms)



Sequential scans (MF) is the baseline.

# Level-aware incremental flattening

$\boxed{\Sigma \vdash^l e \Rightarrow e'}$  In a map-nest context  $\Sigma$ , the source expression  $e$  can be translated at machine level  $l$  into the target expression  $e'$ .

$$\frac{\begin{array}{l} t_{\text{top}}, t_{\text{intra}} \text{ fresh} \qquad \Sigma' = \Sigma, \langle \bar{x} \in \overline{xS} \rangle \\ \Sigma' \vdash_{l+1} e \Rightarrow e_{\text{flat}} \qquad e_{\text{top}} = \mathbf{segmap}^{l+1} \Sigma' e \\ \bullet \vdash_l e \Rightarrow e_{\text{intra}} \qquad e_{\text{middle}} = \mathbf{segmap}^{l+1} \Sigma' e_{\text{intra}} \end{array}}{\Sigma \vdash_{l+1} \mathbf{map} (\lambda \bar{x} \rightarrow e) \overline{xS} \Rightarrow \mathbf{if} \text{Par}(\Sigma') \geq t_{\text{top}} \mathbf{then} e_{\text{top}} \mathbf{else if} \text{Par}(e_{\text{middle}}) \geq t_{\text{intra}} \mathbf{then} e_{\text{middle}} \mathbf{else} e_{\text{flat}}}$$

In the Futhark compiler, only two levels are handled (thread, group), but we believe the idea generalises well.

Representation and Fusion

Handling nested parallelism

Basic flattening rules

Incremental flattening

Multi-level parallelism

**Final words as time permits**

## Block tiling

Level-aware constructs can also be used for expressing other powerful optimisations.

# Block tiling

Level-aware constructs can also be used for expressing other powerful optimisations.



Threads accessing same memory can cooperatively cache it in on-chip memory.

## Motivation for block tiling

```
map (\x -> redomap (+) (\y -> y + x) 0 xs) xs
```

After flattening we get this inner-sequential version:

```
segmapthread ⟨x ∈ xs⟩ (redomap (+) (λy → y + x) 0 xs)
```

**Operation** One thread for each element of xs, and each sequentially traverses xs.

**Problem** ?

## Motivation for block tiling

```
map (\x -> redomap (+) (\y -> y + x) 0 xs) xs
```

After flattening we get this inner-sequential version:

```
segmapthread ⟨x ∈ xs⟩ (redomap (+) (λy → y + x) 0 xs)
```

**Operation** One thread for each element of xs, and each sequentially traverses xs.

**Problem** Poor utilisation of memory bus.

- Many threads simultaneously read same address, which is redundant.
- **Better:** *cooperatively copy block* into on-chip memory and iterate from there.

## Strip mining/chunking the outer segmap

**segmap**<sup>thread</sup>  $\langle x \in xs \rangle$  (**redomap** (+) ( $\lambda y \rightarrow y + x$ ) 0 xs)

Assuming we can split  $xs$  into  $m$  equally sized *tiles* each of size  $t$ , giving  $xss : [m][t] \text{f32}$ , then we can rewrite to

**segmap**<sup>group</sup>  $\langle xs' \in xss \rangle$   
    **segmap**<sup>thread</sup>  $\langle x \in xs' \rangle$   
        **redomap** (+) ( $\lambda y \rightarrow y + x$ ) 0 xs

**Question:** does this compute the same value as the original?



## Strip mining/chunking the outer segmap

$\text{segmap}^{\text{thread}} \langle x \in xs \rangle (\text{redomap } (+) (\lambda y \rightarrow y + x) 0 xs)$

Assuming we can split  $xs$  into  $m$  equally sized *tiles* each of size  $t$ , giving  $xss : [m][t]\text{f32}$ , then we can rewrite to

$\text{segmap}^{\text{group}} \langle xs' \in xss \rangle$   
 $\quad \text{segmap}^{\text{thread}} \langle x \in xs' \rangle$   
 $\quad \quad \text{redomap } (+) (\lambda y \rightarrow y + x) 0 xs$

**Question: does this compute the same value as the original?**

- No—the original expression had type  $[n]\text{f32}$ , while this has type  $[m][t]\text{f32}$
- This can be flattened away.

```

segmapgroup  $\langle xs' \in xss \rangle$ 
  segmapthread  $\langle x \in xs' \rangle$ 
    redomap (+) ( $\lambda y \rightarrow y + x$ ) 0 xs

```

Chunking/strip-mining the **redomap**, we get

```

segmapgroup  $\langle xs' \in xss \rangle$ 
  segmapthread  $\langle x \in xs' \rangle$ 
    loop acc = 0 for ys in xss do
      redomap (+) ( $\lambda y \rightarrow y + x$ ) acc ys

```

```

segmapgroup  $\langle xs' \in xss \rangle$ 
  segmapthread  $\langle x \in xs' \rangle$ 
    redomap (+) ( $\lambda y \rightarrow y + x$ ) 0 xs

```

Chunking/strip-mining the **redomap**, we get

```

segmapgroup  $\langle xs' \in xss \rangle$ 
  segmapthread  $\langle x \in xs' \rangle$ 
    loop acc = 0 for ys in xss do
      redomap (+) ( $\lambda y \rightarrow y + x$ ) acc ys

```

Distributing and interchanging **segmap**<sup>thread</sup> gives

```

segmapgroup  $\langle xs' \in xss \rangle$ 
  loop accs = replicate t 0
  for ys in xss do
    segmapthread  $\langle x, acc \in xs', accs \rangle$ 
      redomap (+) ( $\lambda y \rightarrow y + x$ ) acc ys

```

```

segmapgroup  $\langle xs' \in xss \rangle$ 
  loop accs = replicate t 0
  for ys in xss do
    segmapthread  $\langle x, acc \in xs', accs \rangle$ 
      redomap (+)  $(\lambda y \rightarrow y + x)$  acc ys

```

Collectively copy ys to shared/local memory

```

segmapgroup  $\langle xs' \in xss \rangle$ 
  loop accs = replicate t 0
  for ys in xss do
    let ys' = copy ys in
      segmapthread  $\langle x, acc \in xs', accs \rangle$ 
        redomap (+)  $(\lambda y \rightarrow y + x)$  acc ys'

```

- Now the many iterations of the **redomap** read from fast on-chip memory rather than slower global memory!
- **copy** done collectively by all threads in group

## The fine print

map (\x -> redomap (+) (\y -> y + x) 0 xs) xs  
to

```
segmapgroup <xs' ∈ xss>  
  loop accs = replicate t 0  
  for ys in xss do  
    let ys' = copy ys in  
      segmapthread <x, acc ∈ xs', accs>  
        redomap (+) (λy → y + x) acc ys'
```

- Very simple case (e.g. xss traversed in both loops)
- 2D tiling much more complex
- The *tile size*  $t$  is a sensitive tuning parameter; in this case it should coincide with workgroup size
- Appreciate what a compiler can do for you

# Summary

- There is no *one size fits all*: for optimal performance, we need different amounts of parallelisation for different workloads.
- Incremental flattening generates a *single program* that for varying datasets exploits only as much parallelism as profitable.
- Autotuning for specific hardware and program is needed to select the optimal version at runtime.
- A good IR is as crucial to a compiler as a good language is to a human.