Regular flattening

Troels Henriksen (athas@sigkill.dk)

DIKU
University of Copenhagen

25th of November, 2019

Agenda

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`	*unused*			`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`	*unused*			`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.
- Also affects (internal) language.

”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`:

$$\texttt{sumrows}\ (\texttt{increment}\ a)$$

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

$$\mathbf{map}\ f\ (\mathbf{map}\ g\ a)$$

is *always* equivalent to

$$\mathbf{map}\ (f \circ g)\ a$$

- 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

$$\begin{aligned}
\texttt{sumrows}\ (\texttt{increment}\ a) &= && \text{(Initial expression)}\\
\textbf{map}\ \texttt{sum}\ (\texttt{increment}\ a) &= && \text{(Inline } \texttt{sumrows}\text{)}\\
\textbf{map}\ \texttt{sum}\ (\textbf{map}\ (\lambda r \to \textbf{map}\ (+2)\ r)\ a) &= && \text{(Inline } \texttt{increment}\text{)}\\
\textbf{map}\ (\texttt{sum}\ \circ\ (\lambda r \to \textbf{map}\ (+2)\ r)\ a) &= && \text{(Apply } \textbf{map}\text{-}\textbf{map} \text{ fusion)}\\
\textbf{map}\ (\lambda r \to \texttt{sum}\ (\textbf{map}\ (+2)\ r)\ a) &= && \text{(Apply composition)}
\end{aligned}$$

- 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

$$\overline{z}^{(n)} = z_0, \cdots, z_{(n-1)}$$

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

$$f\ \overline{v}^{(n)} \equiv f\ v_1\ \cdots\ v_n$$

or a tuple

$$(\overline{v}^{(n)}) \equiv (v_1, \ldots, v_n)$$

or a function type

$$\overline{\tau}^{(n)} \to \tau_{n+1} \equiv \tau_1 \to \cdots \to \tau_n \to \tau_{n+1}.$$

For complicated sequences where not all terms under the bar are variant, the variant term is subscripted with *i*.

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

and

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

Fused constructs

Convenient shorthands

$$\texttt{\textbf{redomap}} \odot f\ \overline{d}\ \overline{xs} \equiv \texttt{\textbf{reduce}} \odot \overline{d}\ (\textbf{map}\ f\ \overline{xs})$$

$$\texttt{\textbf{scanomap}} \odot f\ \overline{d}\ \overline{xs} \equiv \texttt{\textbf{scan}} \odot \overline{d}\ (\textbf{map}\ f\ \overline{xs})$$

- 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

$$\mathtt{redomap}\ \odot\ f\ \overline{d}\ \overline{xs} \equiv \mathtt{reduce}\ \odot\ \overline{d}\ (\mathbf{map}\ f\ \overline{xs})$$

$$\mathtt{scanomap}\ \odot\ f\ \overline{d}\ \overline{xs} \equiv \mathtt{scan}\ \odot\ \overline{d}\ (\mathbf{map}\ f\ \overline{xs})$$

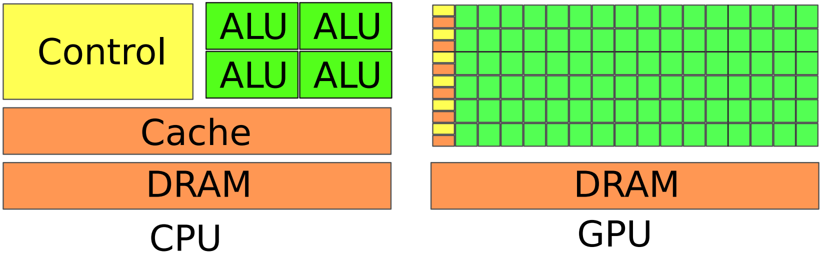
- Emphasises that **reduce**/**scan**-**map** compositions can be considered as a single construct.
- We will see several examples where this is useful.

Note

$$\mathtt{reduce}\ \odot\ \overline{d}\ \overline{xs} \equiv \mathtt{reduce}\ \odot\ \overline{d}\ (\mathbf{map}\ \mathbf{id}\ \overline{xs}) \equiv \mathtt{redomap}\ \odot\ f\ \overline{d}\ \overline{xs}$$

$$\mathtt{scan}\ \odot\ \overline{d}\ \overline{xs} \equiv \mathtt{scan}\ \odot\ \overline{d}\ (\mathbf{map}\ \mathbf{id}\ \overline{xs}) \equiv \mathtt{scanomap}\ \odot\ f\ \overline{d}\ \overline{xs}$$

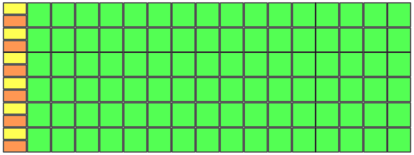
GPUs vs CPUs



- 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*.

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

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?

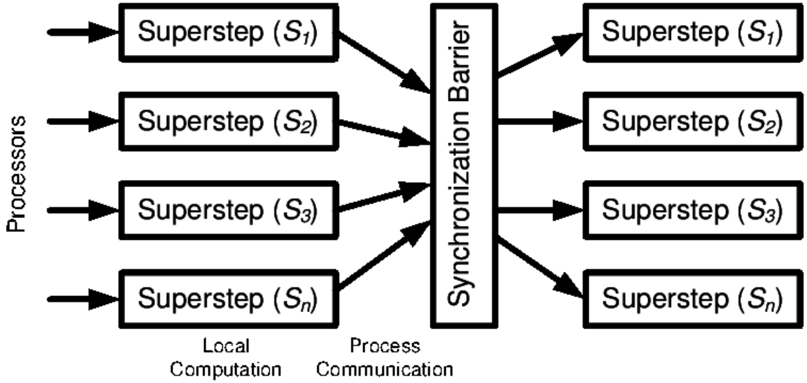
```
mapabs(a) {
  tid = get_thread_id();
  x = a[tid];
  if (x < 0) {
    a[tid] = -x;
  }
}
```

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*:



[1]

- Supersteps are *threads*, which cannot talk to each other.
- The synchronisation barriers are kernel launches.

[1] Illustration by Aftab A. Chandio.

A SOAC-kernel correspondence

The compiler *knows*[2] that certain nests of perfect **map**s correspond to certain GPU basic blocks.

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

Crucial: the **map**s 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.

[2] 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 **map**s 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 **map**s 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
```

$$\Downarrow$$

```
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:

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

[3] *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[3], or just wait until later in the lecture), are applied by the compiler to extract perfect map nests.

[3] *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.

Notation for flat parallelism

Instead of writing

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

We write

$$\textbf{segmap}\ (\langle \mathtt{ps}, \mathtt{rs} \in \mathtt{pss}, \mathtt{rss} \rangle,\ \langle \mathtt{r} \in \mathtt{rs} \rangle,\ \langle \mathtt{p} \in \mathtt{ps} \rangle)$$

$\quad e$

Segmented flat parallel constructs

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

$$\begin{aligned}
\texttt{segmap}\ \Sigma\ e \equiv\ & \texttt{map}\ (\lambda \overline{x_p} \rightarrow \\
& \quad \texttt{map}\ (\lambda \overline{x_{p-1}} \rightarrow \ldots \\
& \qquad \texttt{map}\ (\lambda \overline{x_1} \rightarrow e)\ \overline{y_1}) \\
& \qquad \overline{y_{p-1}}) \\
& \quad \overline{y_p}
\end{aligned}$$

- 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

$$
\begin{aligned}
\mathbf{segred}\ \Sigma\ \odot\ \overline{d}\ e \equiv\ & \mathbf{map}\ (\lambda \overline{x_p} \rightarrow \\
& \quad \mathbf{map}\ (\lambda \overline{x_{p-1}} \rightarrow \ldots \\
& \quad\quad \mathbf{redomap}\ \odot\ (\lambda \overline{x_1} \rightarrow e)\ \overline{d}\ \ \overline{y_1}) \\
& \quad\quad \overline{y_{p-1}}) \\
& \quad \overline{y_p}
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{segscan}\ \Sigma\ \odot\ \overline{d}\ e \equiv\ & \mathbf{map}\ (\lambda \overline{x_p} \rightarrow \\
& \quad \mathbf{map}\ (\lambda \overline{x_{p-1}} \rightarrow \ldots \\
& \quad\quad \mathbf{scanomap}\ \odot\ (\lambda \overline{x_1} \rightarrow e)\ \overline{d}\ \overline{y_1}) \\
& \quad\quad \overline{y_{p-1}}) \\
& \quad \overline{y_p}
\end{aligned}
$$

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

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

$$\frac{}{\mathcal{V} \vdash e_1 \Rightarrow e_2} \qquad \frac{}{\mathcal{V} \vdash v \Rightarrow \mathcal{V}(v)} \qquad \frac{\mathcal{V} \vdash e_1 \Rightarrow \texttt{true}}{\mathcal{V} \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \Rightarrow e_2}$$

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.

$$\frac{}{\mathcal{V} \vdash e_1 \Rightarrow e_2} \qquad \frac{}{\mathcal{V} \vdash v \Rightarrow \mathcal{V}(v)} \qquad \frac{\mathcal{V} \vdash e_1 \Rightarrow \texttt{true}}{\mathcal{V} \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Rightarrow e_2}$$

$$\frac{\mathcal{V}, x \mapsto e_1 \vdash e_2 \Rightarrow e_2'}{\mathcal{V} \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \Rightarrow \textbf{let } x = e_1 \textbf{ in } e_2'} \qquad \frac{x \notin FV(e_2)}{\mathcal{V} \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \Rightarrow e_2}$$

- 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 Σ, the source expression e can be translated into the target expression e'.

$$\frac{e \text{ has inner SOACs} \qquad \Sigma, \langle \overline{x} \in \overline{\mathtt{xs}} \rangle \vdash e \Rightarrow e_{\mathrm{flat}}}{\Sigma \vdash \mathbf{map}\ (\lambda \overline{x} \to e)\ \overline{\mathtt{xs}} \Rightarrow e_{\mathrm{flat}}}$$

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

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

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

Rule for map distribution

$$\frac{\begin{array}{c}\text{size of each array in } \overline{a_0} \text{ invariant to } \Sigma \\ \Sigma = \langle \overline{x_p} \in \overline{y_p} \rangle, \ldots, \langle \overline{x_1} \in \overline{y_1} \rangle \qquad \Sigma \vdash e_1 \Rightarrow e_1' \\ \overline{a_p}, \ldots, \overline{a_1} \text{ fresh names} \qquad \Sigma' \vdash e_2 \Rightarrow e_2' \\ \Sigma' = \langle \overline{x_p}\, \overline{a_{p-1}} \in \overline{y_p}\, \overline{a_p} \rangle, \ldots, \langle \overline{x_1}\, \overline{a_0} \in \overline{y_1}\, \overline{a_1} \rangle \end{array}}{\Sigma \vdash \texttt{let}\ \overline{a_0} = e_1\ \texttt{in}\ e_2 \Rightarrow \texttt{let}\ \overline{a_p} = e_1'\ \texttt{in}\ e_2'}$$

Rule for map distribution

$$\frac{\begin{array}{c}\text{size of each array in } \overline{a_0} \text{ invariant to } \Sigma \\ \Sigma = \langle \overline{x_p} \in \overline{y_p} \rangle, \ldots, \langle \overline{x_1} \in \overline{y_1} \rangle \qquad \Sigma \vdash e_1 \Rightarrow e_1' \\ \overline{a_p}, \ldots, \overline{a_1} \text{ fresh names} \qquad \Sigma' \vdash e_2 \Rightarrow e_2' \\ \Sigma' = \langle \overline{x_p}\,\overline{a_{p-1}} \in \overline{y_p}\,\overline{a_p} \rangle, \ldots, \langle \overline{x_1}\,\overline{a_0} \in \overline{y_1}\,\overline{a_1} \rangle\end{array}}{\Sigma \vdash \textbf{let } \overline{a_0} = e_1 \textbf{ in } e_2 \Rightarrow \textbf{let } \overline{a_p} = e_1' \textbf{ in } e_2'}$$

Example for

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

Suppose already inside the outer map

$\Sigma = \langle \mathtt{xs} \in \mathtt{xss} \rangle \quad \Sigma' = \langle \mathtt{xs}, \mathtt{y} \in \mathtt{xss}, \mathtt{ys} \rangle$

$\Sigma \vdash \mathtt{redomap\ (+)\ (}\lambda\mathtt{x\ \to\ x)\ 0\ xs} \Rightarrow \textbf{segred}\,(\Sigma, \langle \mathtt{x} \in \mathtt{xs} \rangle)\ (+)\ 0\ \mathtt{x}$

$\Sigma' \vdash \mathtt{map\ (}\lambda\mathtt{x\ \to\ x\ +\ y)\ xs} \Rightarrow \textbf{segmap}\,(\Sigma', \langle \mathtt{x} \in \mathtt{xs} \rangle)\ \mathtt{x\ +\ y}$

$\Sigma \vdash \ldots \Rightarrow \quad \textbf{let}\ \mathtt{ys} = \textbf{segred}\,(\langle \mathtt{xs} \in \mathtt{xss} \rangle, \langle \mathtt{x} \in \mathtt{xs} \rangle)\ (+)\ 0\ \mathtt{x}$
$\qquad\qquad\ \textbf{in segmap}\ \langle \mathtt{xs}, \mathtt{y} \in \mathtt{xss}, \mathtt{ys} \rangle\ \mathtt{x\ +\ y}$

Handling transposition

`rearrange` $(d_1, \cdots, d_n)$ 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 $d_1, \cdots, d_n$. E.g:

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

Handling transposition

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

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

Flattening rule

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

Handling transposition

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

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

Flattening rule

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

Example

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

Handling transposition

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

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

Flattening rule

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

Example

$$\bullet \vdash \texttt{\textbf{map}}\ (\lambda \mathtt{x} \rightarrow \texttt{\textbf{rearrange}}\ (1, 0)\ \mathtt{x})\ \mathtt{xs} \Rightarrow \texttt{\textbf{rearrange}}\ (0, 2, 1)\ \mathtt{xs}$$

Moderate flattening

Nondeterministic flattening rules do not specify an *algorithm*. The Futhark compiler applies them using various heuristics.

- Nested **`scanomap`**s are always parallelised.
- Nested **`redomap`**s are never parallelised, *unless* they correspond exactly to a **`reduce`**.
- **`loop`**s are always interchanged if possible.
- Distribution is only done when necessary to isolate constructs for parallelisation.

Moderate flattening

Nondeterministic flattening rules do not specify an *algorithm*. The Futhark compiler applies them using various heuristics.

- Nested **`scanomap`**s are always parallelised.
- Nested **`redomap`**s are never parallelised, *unless* they correspond exactly to a **`reduce`**.
- **`loop`**s are always interchanged if possible.
- Distribution is only done when necessary to isolate constructs for parallelisation.

Example—for this program, only the outermost map is parallelised.

```
map (\x -> reduce (+) 0 (map (+x) ys)) xs
```

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
```

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

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 the outer* **map***s do not saturate the 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 the outer* **maps** *do not saturate the GPU*

Moderate flattening

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

- **Only outer parallelism**
- The **redomap** can then be block tiled
- *Best if the outer* **maps** *saturate the GPU*

There is no *one size fits all*—and 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.
- **Moderate flattening**—as discussed previously—uses a compile-time heuristic to pick between these options.
- **Incremental flattening** generates *both* versions and uses a predicate to pick at runtime.

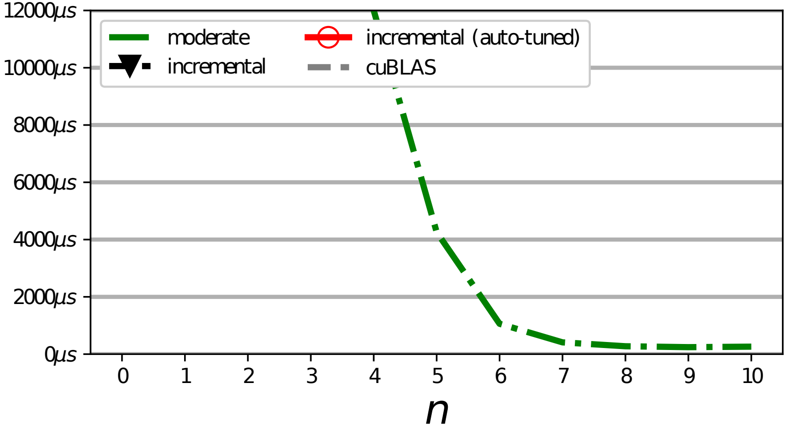
Multi-versioned matrix multiplication

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

```
if n * m > $t_0$ then
  map (\xs ->
         map (\ys ->
                redomap (+) (*) 0 xs ys)
             (transpose yss))
      xss
else
  map (\xs ->
         map (\ys ->
                redomap (+) (*) 0 xs ys)
             (transpose yss))
      xss
```

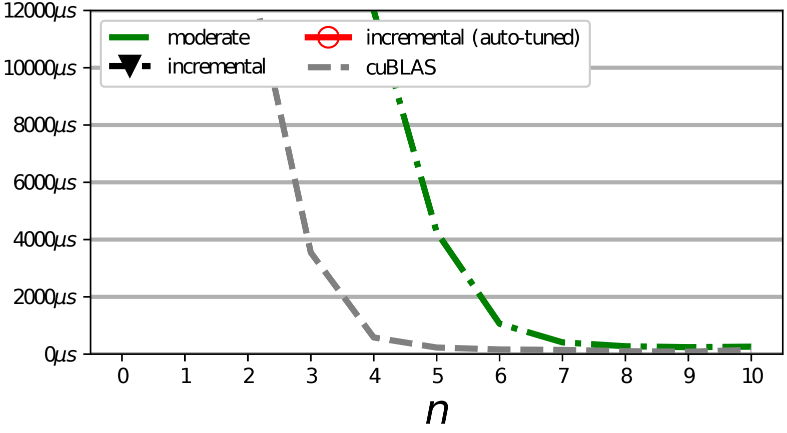
The t_0 *threshold parameter* is used to select between the two versions—and should be auto-tuned on the concrete hardware.

Matrix multiplication on NVIDIA K40



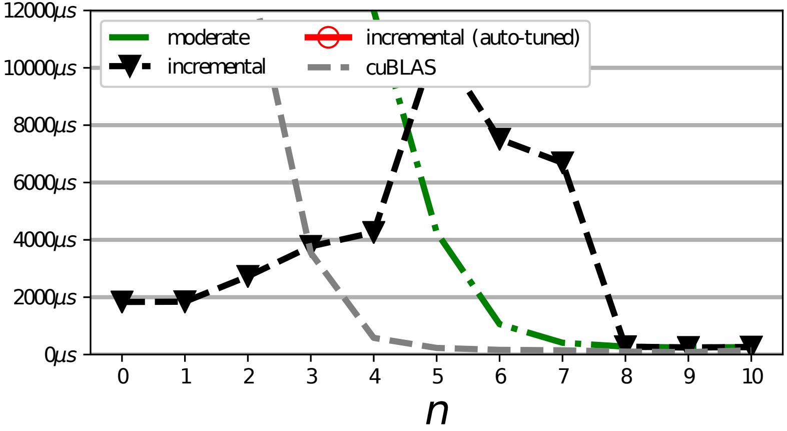
Multiplying matrices of size $2^n \times 2^m$ and $2^m \times 2^n$, where $m = 25 - 2n$, meaning that work is constant as we vary n.

Matrix multiplication on NVIDIA K40



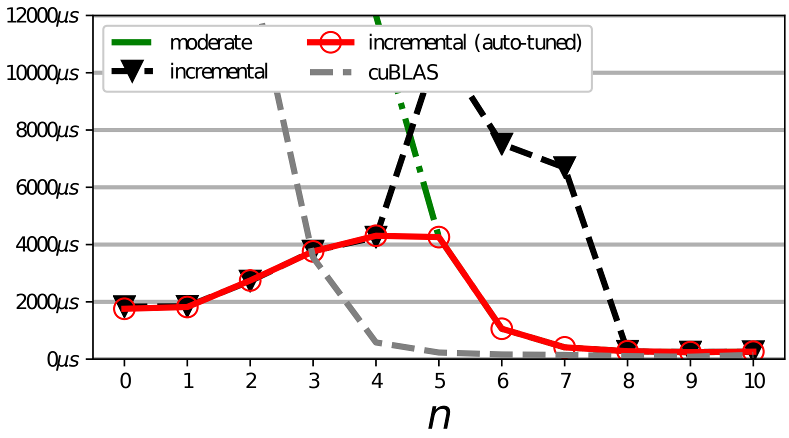
Multiplying matrices of size $2^n \times 2^m$ and $2^m \times 2^n$, where $m = 25 - 2n$, meaning that work is constant as we vary n.

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

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

Example for

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

Incremental flattening rule

$$\frac{\Sigma' = \Sigma, \langle \overline{x} \in \overline{\mathtt{xs}} \rangle \quad \Sigma' \vdash e \Rightarrow e_{\mathrm{flat}}}{\begin{array}{l}\Sigma \vdash \textbf{map}\ (\lambda \overline{x} \rightarrow e)\ \overline{\mathtt{xs}} \Rightarrow \\ \qquad \mathtt{\textbf{if}}\ \mathrm{Par}(\Sigma') \geq t_{\mathrm{top}} \\ \qquad \mathtt{\textbf{then}}\ \textbf{segmap}\ \Sigma'\ e \\ \qquad \mathtt{\textbf{else}}\ e_{\mathrm{flat}}\end{array}}$$

Example for

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

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

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

$\Sigma \vdash \ldots \Rightarrow$

$\mathtt{\textbf{if}}\ \mathrm{length}(\mathtt{xss}) \geq t_{\mathrm{top}}$
$\mathtt{\textbf{then}}\ \textbf{segmap}\ \langle \mathtt{xs} \in \mathtt{xss} \rangle\ (\mathtt{redomap\ (+)\ (\lambda x \rightarrow x)\ 0\ xs})$
$\mathtt{\textbf{else}}\ \textbf{segred}\ (\langle \mathtt{xs} \in \mathtt{xss} \rangle, \langle \mathtt{x} \in \mathtt{xs} \rangle)\ (+)\ 0\ \mathtt{x}$

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.

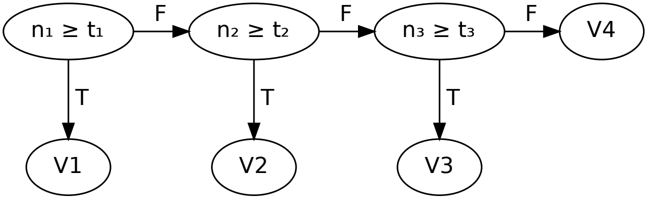
A *configuration* P maps each t_i to an integer $P(t_i)$.

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[4]



- 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.

[4]https://futhark-lang.org/student-projects/svend-msc-thesis.pdf

Using incremental flattening

Set the environment variable `FUTHARK_INCREMENTAL_FLATTENING=1` before running the compiler or other tools (`futhark bench` etc).

```
$ export FUTHARK_INCREMENTAL_FLATTENING=1
$ futhark opencl matmul.fut
```

To autotune:

```
$ export FUTHARK_INCREMENTAL_FLATTENING=1
$ futhark autotune --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 --kernels matmul.fut` to see IR.

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.

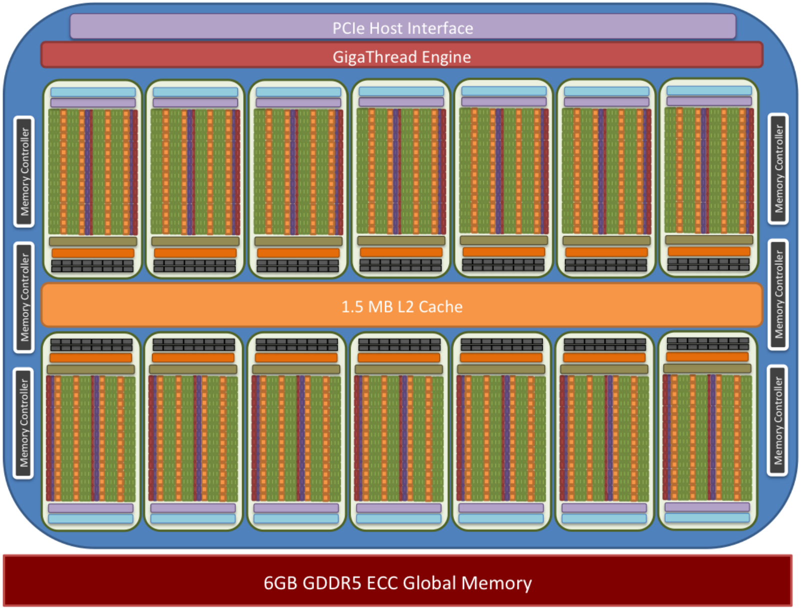
- Most hardware has useful (fixed) levels of parallelism.
- An ideal flattening algorithm maps levels of application parallelism (any number) to hardware parallelism (fixed number) in a way that exploits locality well.

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.

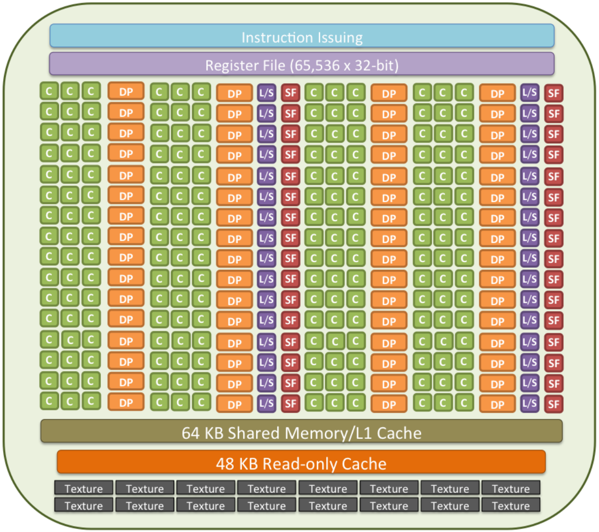
General principle

“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

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

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

$$\mathbf{segmap}^l\ \Sigma\ e$$
$$\mathbf{segscan}^l\ \Sigma\ \odot\ \overline{d}\ e$$
$$\mathbf{segred}^l\ \Sigma\ \odot\ \overline{d}\ e$$

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

$$\mathbf{segmap}^{\text{thread}}\ \langle \mathtt{x} \in \mathtt{xs} \rangle\ (\mathtt{transpose\ x})$$

Each workgroup transposes part of an array

$$\mathbf{segmap}^{\text{group}}\ \langle \mathtt{x} \in \mathtt{xs} \rangle\ (\mathtt{transpose\ x})$$

These are both equivalent to `map transpose xs`.

Each workgroup sums the row of an array

$$\mathbf{segmap}^{\text{group}}\ \langle \mathtt{xs} \in \mathtt{xss} \rangle\ (\mathbf{segred}^{\text{thread}}\ \langle \mathtt{x} \in \mathtt{xs} \rangle\ (+)\ 0\ \mathtt{x})$$

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`

$$\textbf{segmap}^{\text{thread}}\ (\langle \mathtt{xss} \in \mathtt{xssss}\rangle, \langle \mathtt{xs} \in \mathtt{xss}\rangle)$$

```
 let bs = scan ⊕ $d_\oplus$ xs
 let cs = scan ⊗ $d_\otimes$ bs
 in scan ⊙ $d_\odot$ $cs$
```

`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

Moderate and incremental 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 =
  segscan$^{\text{thread}}$ $(\langle \mathtt{xss} \in \mathtt{xsss}\rangle, \langle \mathtt{xs} \in \mathtt{xss}\rangle, , \langle \mathtt{x} \in \mathtt{xs}\rangle)\ \oplus\ d_{\oplus}\ \mathtt{x}$
let csss =
  segscan$^{\text{thread}}$ $(\langle \mathtt{bss} \in \mathtt{bsss}\rangle, \langle \mathtt{bs} \in \mathtt{bss}\rangle, , \langle \mathtt{b} \in \mathtt{bs}\rangle)\ \oplus\ d_{\oplus}\ \mathtt{b}$
in
  segscan$^{\text{thread}}$ $(\langle \mathtt{css} \in \mathtt{csss}\rangle, \langle \mathtt{cs} \in \mathtt{css}\rangle, , \langle \mathtt{c} \in \mathtt{cs}\rangle)\ \oplus\ d_{\oplus}\ \mathtt{c}$
```

This is what full and moderate flattening will do.

Option III: Mapping innermost parallelism to the workgroup level

```
map (\xss ->
      map (\xs ->
             let bs = scan ⊕ d_⊕ xs
             let cs = scan ⊗ d_⊗ bs
             in  scan ⊙ d_⊙ cs)
          xss)
    xsss
```

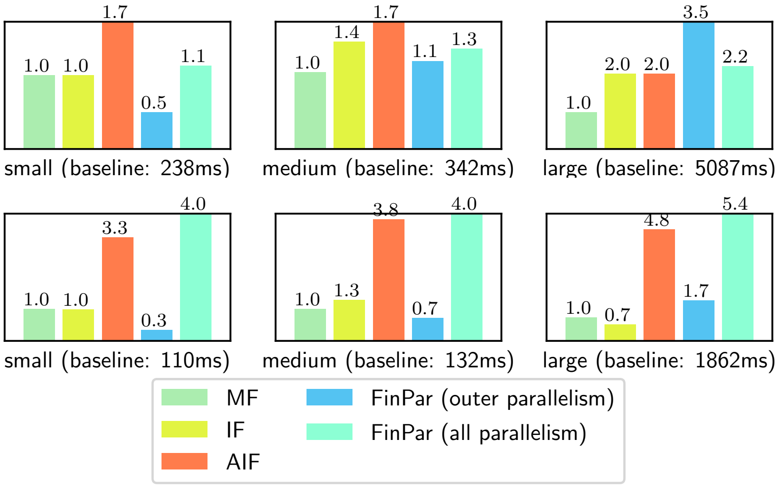
Option III: Mapping innermost parallelism to the workgroup level

$$\mathbf{segmap}^{\text{group}}\ (\langle \mathtt{xss} \in \mathtt{xsss}\rangle, \langle \mathtt{xs} \in \mathtt{xss}\rangle)$$
$$\begin{aligned}
&\mathbf{let}\ \mathtt{bs} = \mathbf{segscan}^{\text{thread}}\ \langle \mathtt{x} \in \mathtt{xs}\rangle\ \oplus\ d_{\oplus}\ \mathtt{x}\\
&\mathbf{let}\ \mathtt{cs} = \mathbf{segscan}^{\text{thread}}\ \langle \mathtt{b} \in \mathtt{bs}\rangle\ \otimes\ d_{\otimes}\ \mathtt{b}\\
&\mathbf{in}\ \mathbf{segscan}^{\text{thread}}\ \langle \mathtt{c} \in \mathtt{cs}\rangle\ \otimes\ d_{\otimes}\ \mathtt{c}
\end{aligned}$$

- Iterations of outer **segmap**s assigned to GPU workgroups[5]
- Each $\mathbf{segscan}^{\text{thread}}$ is executed collaboratively by a workgroup and in local memory[6]
- Only works if the innermost parallelism fits in a workgroup
- In our implementation, the default threshold parameters (almost) never pick this, so auto-tuning is required

[5] *Thread block* in CUDA
[6] *Shared memory* in CUDA

LocVolCalib performance (higher is better)



Speedup for LocVolCalib on NVIDIA K40 (top) and AMD Vega 64 (bottom). Moderate flattening is the baseline.

Level-aware incremental flattening

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

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

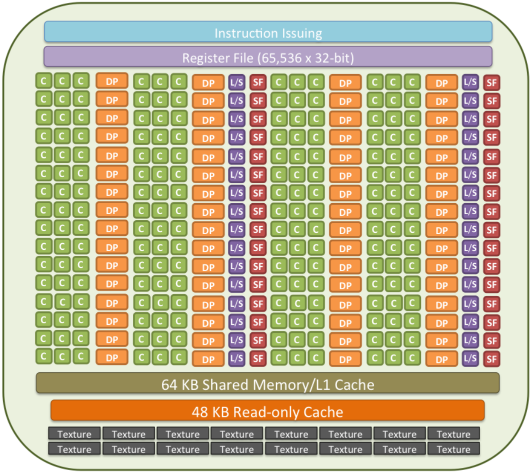
In the Futhark compiler, only two levels are handled (thread, group), but we believe the idea generalises well.

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 cooperate in caching it in on-chip local/shared memory.

Motivation for block tiling

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

After (moderate) flattening we get

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

Operation	One thread for each element of `xs`, each of which sequentially traverses `xs`
Problem	?

Motivation for block tiling

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

After (moderate) flattening we get

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

Operation	One thread for each element of `xs`, each of which sequentially traverses `xs`
Problem	Poor utilisation of memory bus.

- Many threads are simultaneously reading the same part of memory, which is redundant.
- Better to *cooperatively* read an entire *block* into on-chip memory and iterate from there.

Strip mining/chunking the outer `segmap`

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

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

```
segmap^group ⟨xs' ∈ xss⟩
  segmap^thread ⟨x ∈ xs'⟩
    redomap (+) (λy → y + x) 0 xs
```

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

Strip mining/chunking the outer segmap

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

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

```
segmap^group ⟨xs' ∈ xss⟩
  segmap^thread ⟨x ∈ xs'⟩
    redomap (+) (λy → y + x) 0 xs
```

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

No—the original expression had type `[n]f32`, while this has type `[m][t]f32`. This can be `flattened` away.

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

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

```
  **segmap**$^{\text{group}}$ $\langle$xs' $\in$ xss$\rangle$
    **segmap**$^{\text{thread}}$ $\langle$x $\in$ xs'$\rangle$
      **loop** acc = 0 **for** ys **in** xss **do**
        **redomap** $(+)$ $(\lambda$y $\rightarrow$ y $+$ x$)$ acc ys
```

```
segmap^group ⟨xs' ∈ xss⟩
  segmap^thread ⟨x ∈ xs'⟩
    redomap (+) (λy → y + x) 0 xs
```

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

```
segmap^group ⟨xs' ∈ xss⟩
  segmap^thread ⟨x ∈ xs'⟩
    loop acc = 0 for ys in xss do
      redomap (+) (λy → y + x) acc ys
```

Distributing and interchanging **segmap**$^{\text{thread}}$ gives

```
segmap^group ⟨xs' ∈ xss⟩
  loop accs = replicate t 0
  for ys in xss do
   segmap^thread ⟨x, acc ∈ xs', accs⟩
     redomap (+) (λy → y + x) acc ys
```

```
segmap^group ⟨xs' ∈ xss⟩
  loop accs = replicate t 0
  for ys in xss do
    segmap^thread ⟨x, acc ∈ xs', accs⟩
      redomap (+) (λy → y + x) acc ys
```

Collectively copy ys to shared/local memory

```
segmap^group ⟨xs' ∈ xss⟩
  loop accs = replicate t 0
  for ys in xss do
    let ys' = copy ys in
      segmap^thread ⟨x, acc ∈ xs', accs⟩
        redomap (+) (λy → 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

```
segmap^group ⟨xs' ∈ xss⟩
   loop accs = replicate t 0
   for ys in xss do
      let ys' = copy ys in
        segmap^thread ⟨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 absolutely crucial for a compiler

To use incremental flattening, set the environment variable `FUTHARK_INCREMENTAL_FLATTENING=1` before running the compiler.