Automatic SIMD vectorization for Haskell

Leaf Petersen, **Dominic Orchard**, Neal Glew

ICFP 2013 - Boston, MA, USA

Work done at Intel Labs

SIMD

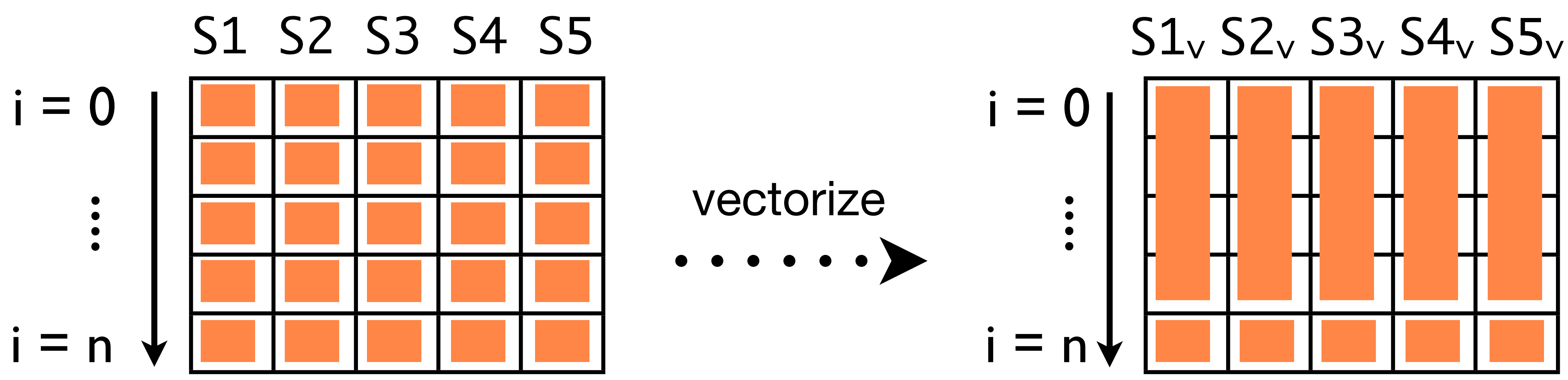
- Trend towards parallel architectures (multi-core & instruction-level parallelism)
- SIMD: fine-grained data parallelism
    - Vector registers: e.g. 128-bit wide (4x integers)

A_0	A_1	A_2	A_3

    - Vector instructions: e.g. `vadd r1, r2`

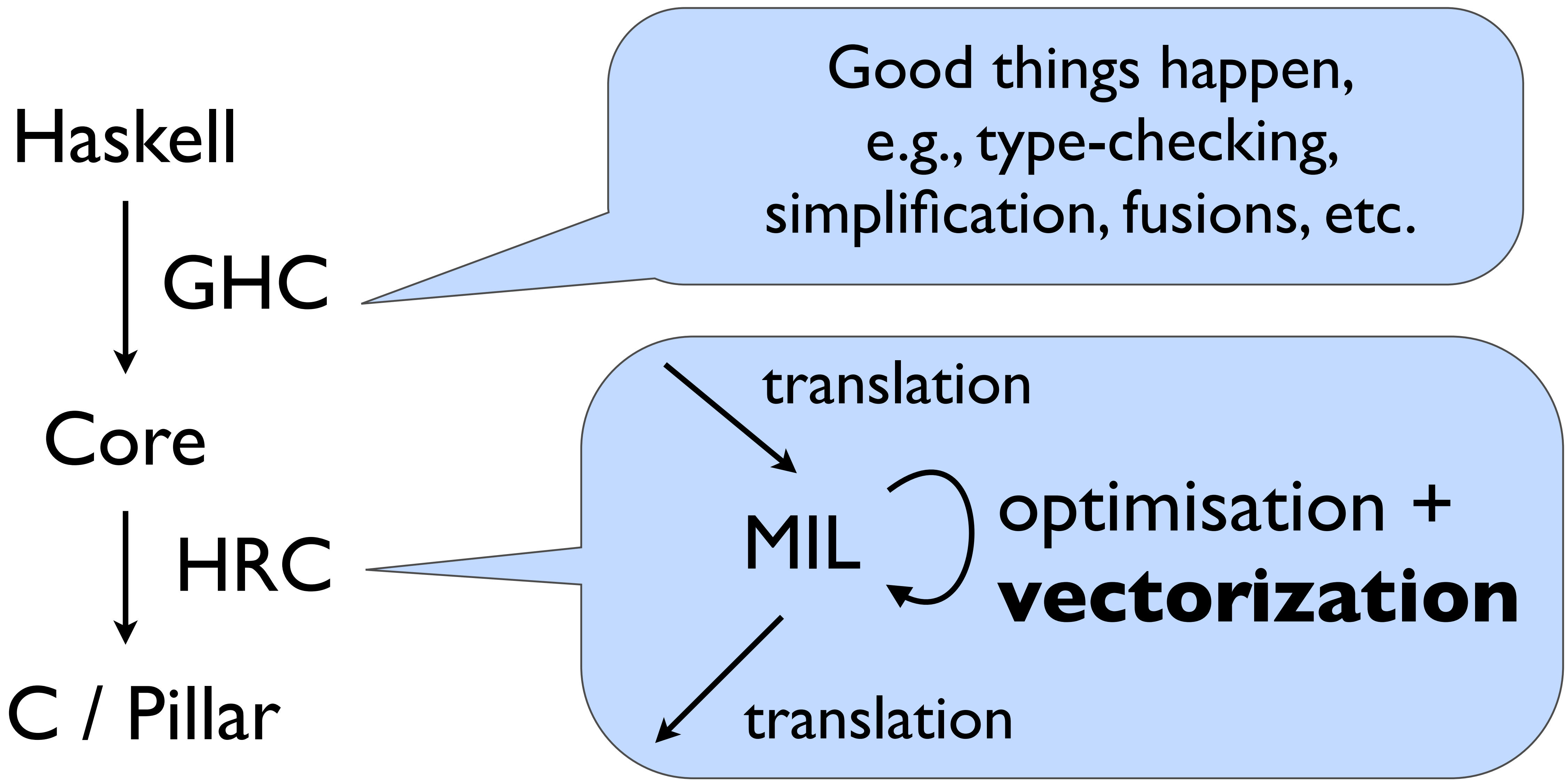
$r1 =$	A_0	A_1	A_2	A_3
$r2 =$	B_0	B_1	B_2	B_3
`vadd r1, r2` ⇢ $r1 =$	A_0+B_0	A_1+B_1	A_2+B_2	A_3+B_3

Exploiting SIMD



- Need to preserve semantics
- Imperative: (undecidable) dependency & effects analysis
- Functional: much easier

Intel Labs Haskell Research Compiler (HRC)



[1] Liu, Glew, Peterson, Anderson, *The Intel Labs Haskell Research Compiler,* Haskell '13, ACM

Take home messages

- FP optimisation not exhausted: still low-hanging fruit to be had
- Vectorization is low hanging and a big win:
  - up to 6.5x speedups for HRC + vectorization
- Many standard techniques + a few extras

MIL

- Aimed at compiling high-performance functional code
- Block-structured (CFG) (loops), SSA form
- Strict + explicit thunks
- Distinguishes mutable and immutable data
- Vector primitives (numerical and array ops)

Prior to vectorization

- Core → MIL: closure conversion, explicit thunks
- Optimisations (general simplifier, *unboxing*, representation opts.)
- *Contification* [1]
  - Converts (many) tail recursion uses to loops

[1] M. Fluet and S. Weeks. Contification using dominators. ICFP 2001, ACM.

Vectorization

- Targets inner-most loops that are:
  - reductions over immutable arrays
  - *initialising writes*

Initialising writes

- Allocate then initialize an (immutable) array
- Two invariants:
  - Reading an array element always follows the initializing write of the element
  - Each element may be initialized only once
- Modified GHC libraries to generate <u>intializing writes</u> rather than <u>mutation</u>

Mutability in library (Data.Vector)

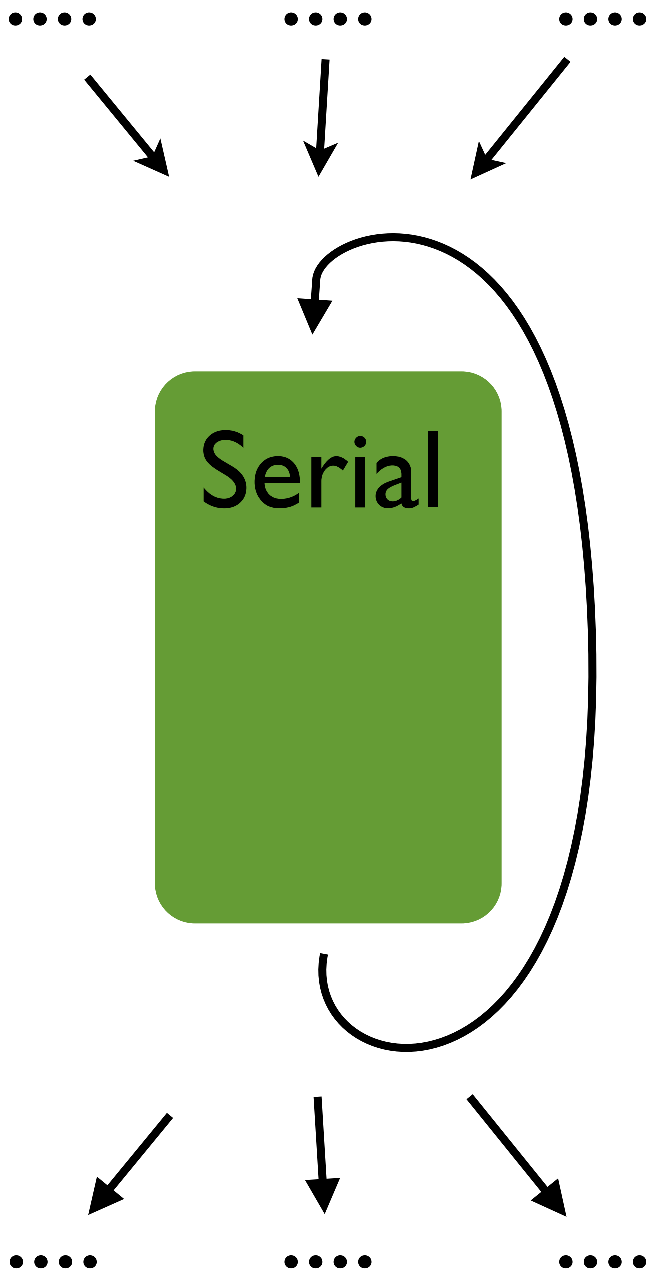
```
unstreamRMax
  :: (PrimMonad m, MVector v a) => MStream m a -> Int -> m (v
(PrimState m) a)
unstreamRMax s n
  = do
      v <- INTERNAL_CHECK(checkLength) "unstreamRMax" n
           $ unsafeNew n
      let put i x = do
                      let i' = i-1
                      INTERNAL_CHECK(checkIndex) "unstreamRMax" i' n
                        $ unsafeWrite v i' x
                      return i'
      i <- MStream.foldM' put n s
      return $ INTERNAL_CHECK(checkSlice) "unstreamRMax" i (n-i) n
             $ unsafeSlice i (n-i) v
```

(used for the immutable vector too)

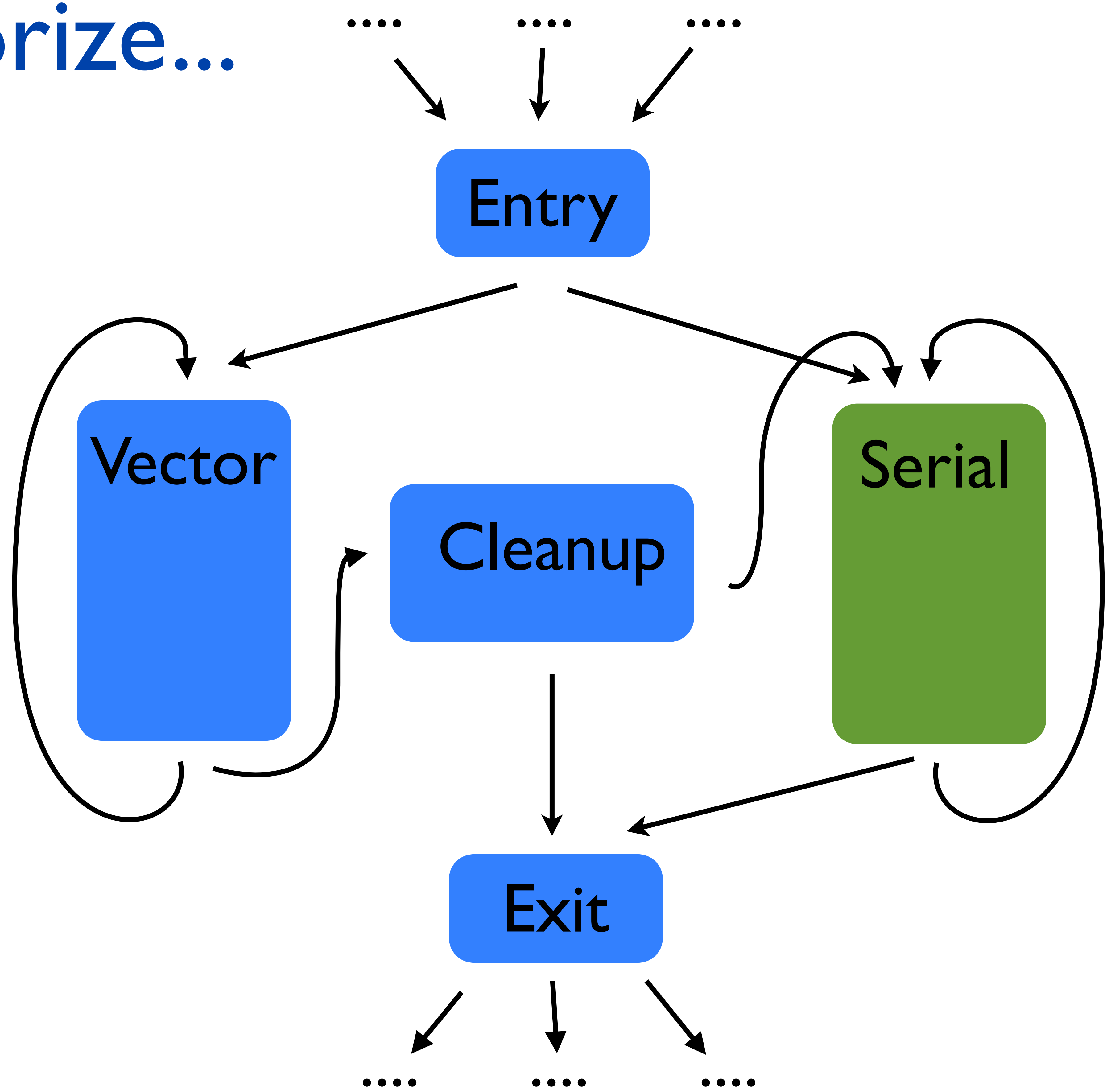
Transformed to immutability + initializing writes

```
unstreamRPrimMmax :: (PrimMonad m, Vector v a) => Int -> MStream m a
-> m (v a)
unstreamRPrimMmax n s = do
  v <- INTERNAL_CHECK(checkLength) "unstreamRPrimMmax" n
       $ unsafeCreate n
  let put i x = do
                 let i' = i-1
                 INTERNAL_CHECK(checkIndex) "unstreamMmax" i' n
                   $ unsafeInitElem v i' x
                 return i'
  i <- MStream.foldM' put n s
  v' <- basicUnsafeInited v
  return $ INTERNAL_CHECK(checkSlice) "unstreamRPrimMmax" i (n-i) n
         $ unsafeSlice i (n-i) v'
```

Start with...



Vectorize...



Vectorization

- Transform each instruction
  - ... depending on properties of instruction/arguments
- Let dead-code elimination handle clean up

$x = \ldots.$

vectorize $\cdots\cdots\blacktriangleright$

$x^s = \ldots.$	first value (scalar)
$x^v = \ldots.$	vector version
$x^l = \ldots.$	last value (scalar)
($x^b = \ldots.$	basis vector)

Vectorization (simple)

- Pointwise operations, promote all constants

$$y = +(x, 4) \longrightarrow y^{v} = \langle + \rangle(x^{v}, \langle 4, 4, 4, 4 \rangle)$$

pointwise op

vector-promoted constant

projection

$$y^{l} = y^{v} \,!\, 3$$

last value of y, needed if y is live-out

Vectorization (simple)

- Pointwise operations, promote all constants

$$y = x[z] \longrightarrow y^v = \langle x^v \rangle[\langle z^v \rangle]$$

general array load on vectors "gather"

equivalent to:

$$y^v = \langle x^v{}_0[z^v{}_0],\ x^v{}_1[z^v{}_1],\ x^v{}_2[z^v{}_2],\ x^v{}_3[z^v{}_3] \rangle$$

$$x[z] = y \longrightarrow \langle x^v \rangle[\langle z^v \rangle] = y^v$$

general array store on vectors "scatter"

Why is this (sometimes) naive?

- General vector array loads/stores not widely supported
- Specialised versions often faster
- Dependency between scalar/vector
- Does too much work (non-optimal)
    - HRC instead tracks "induction variables" and uses this information to optimise

Induction variables

- Base induction variables
  - Loop-carried variable with constant step
- Derived induction variable
  - Induction variable $+$ constant or $\times$ constant

$L_1(x)$: ....

  $x' = +(x, 1)$

if ... goto $L_1(x')$ else goto $L_{end}(...)$

x is a base induction variable (step 1)

x' is a derived induction variable

Vectorizing base I.V.s e.g. $\texttt{x}$

$L_1(x)$: ....
  $x' = +(x, 1)$
if ... goto $L_1(x')$ else goto $L_{end}(...)$

HRC:

$x^b = \langle 0, 1, 2, 3 \rangle$ — *basis* vector

$x^v = \langle + \rangle(x^b, \langle x^s, x^s, x^s, x^s \rangle)$

$x^l = x^s + 3$ — *last* basis value

Vectorizing derived I.V.s e.g. x'

$L_1(x)$: ....
  $x' = +(x, 1)$
if ... goto $L_1(x')$ else goto $L_{end}(...)$

Simple:

$x^b = \langle 0, 1, 2, 3 \rangle$

$x^v = \langle + \rangle(x^b, \langle x^s, x^s, x^s, x^s \rangle)$

$x'^v = \langle + \rangle(x^v, \langle 1, 1, 1, 1 \rangle)$

2 vector ops
+ 1 promotions

HRC:

$x'^b = x^b$

$x'^s = x^s + 1$

$x'^v = \langle + \rangle(x'^b, \langle x'^s, x'^s, x'^s, x'^s \rangle)$

Removes dependence on x^v
1 vector op
+ 1 scalar op
+ 1 promotion

$x'^l = x^l + 1$

Vectorizing induction variables

$$x'^{v} = \langle + \rangle(x^{v}, \langle 1, 1, 1, 1 \rangle)$$ [simple approach]

$$= \langle + \rangle(\langle + \rangle(x^{b}, \langle x^{s}, x^{s}, x^{s}, x^{s} \rangle), \langle 1, 1, 1, 1 \rangle)$$

$$= \langle + \rangle(x^{b}, \langle + \rangle(\langle x^{s}, x^{s}, x^{s}, x^{s} \rangle, \langle 1, 1, 1, 1 \rangle))$$ (assoc)

"Naturality" of *promotion:*

$$\langle\, f \,\rangle \;.\; (promote \times promote) \equiv promote \;.\; f$$

$$\langle f(a,b), f(a,b), f(a,b), f(a,b) \rangle \equiv \langle f \rangle(\langle a, a, a, a \rangle, \langle b, b, b, b \rangle)$$

$$= \langle + \rangle(x^{b}, \langle x^{s} + 1\,, x^{s} + 1, x^{s} + 1, x^{s} + 1 \rangle))$$ (naturality)

$$= \langle + \rangle(x^{b}, \langle x'^{s}, x'^{s}, x'^{s}, x'^{s} \rangle))$$ (simplify)

Vectorizing loads/stores

- Specialised gathers and scatters for unit strides provide higher performance

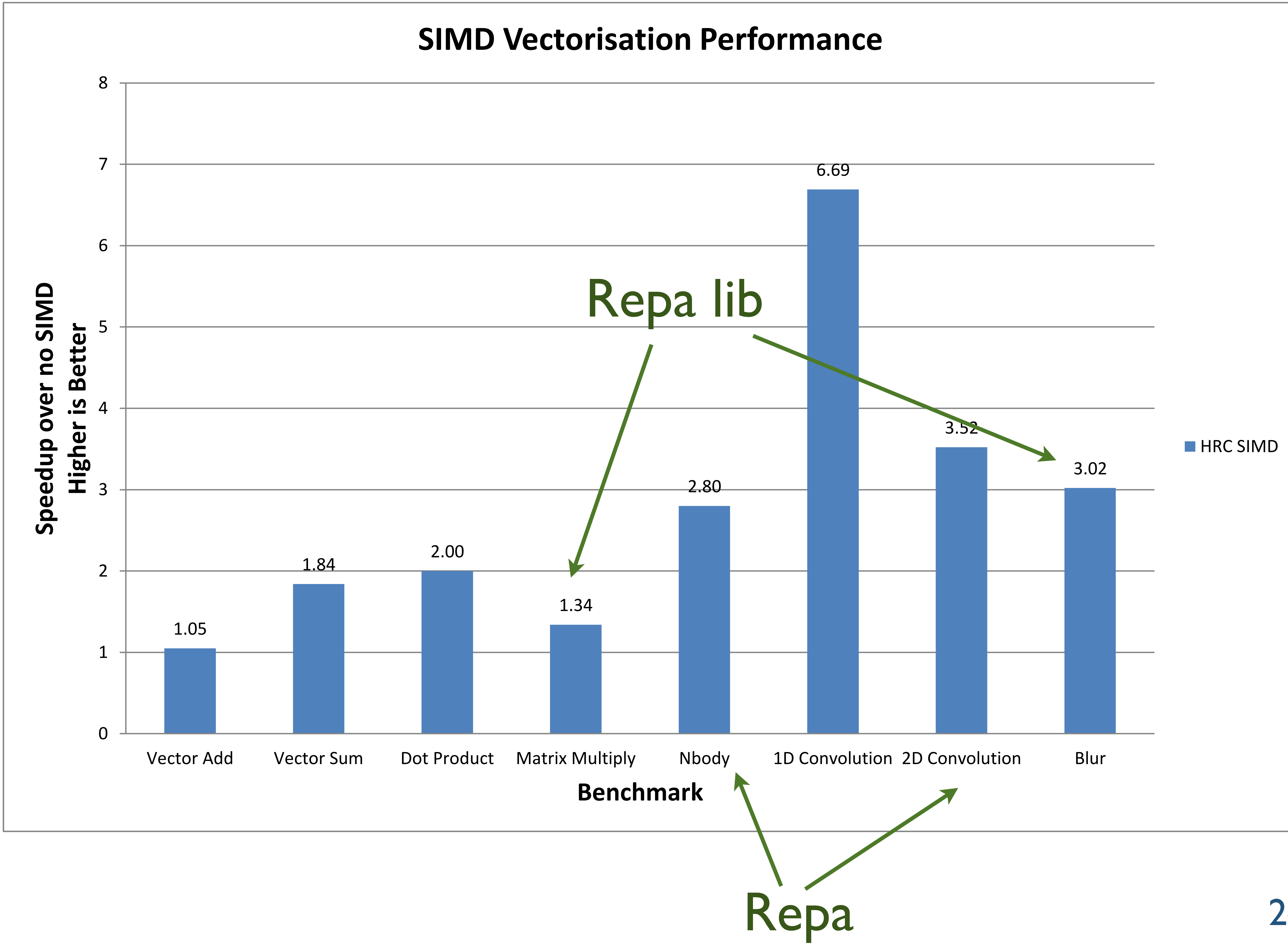
specialised (*contiguous*) array load

$$\mathrm{y} = \mathrm{x}[\mathrm{z}] \longrightarrow \mathrm{y}^{\mathrm{v}} = \mathrm{x}^{\mathrm{s}}[\langle \mathrm{z}^{\mathrm{s}}\!:\rangle]$$

(when x is loop invariant, z is a unit stride induction variable)

Results

Test framework: Intel Xeon, 256-bit register AVX



SIMD Vectorisation Performance
Speedup over no SIMD
Higher is Better
8
7
6
5
4
3
2
1
0
1.05
1.84
2.00
1.34
2.80
6.69
3.52
3.02
Vector Add
Vector Sum
Dot Product
Matrix Multiply
Nbody
1D Convolution
2D Convolution
Blur
Benchmark
HRC SIMD
Repa lib
Repa

Conclusions: what is important?

- Purity at the top level
- Fusion
- Understanding effects at the implementation level
- Use program properties (induction vars)
- Keep dependencies between scalars/vectors separate

we already knew these (the first two items)

Conclusions

- Future work
  - Masked instructions
  - Vectorised allocations
  - Alignment

- SIMD was straightforward to add to HRC, with very good results

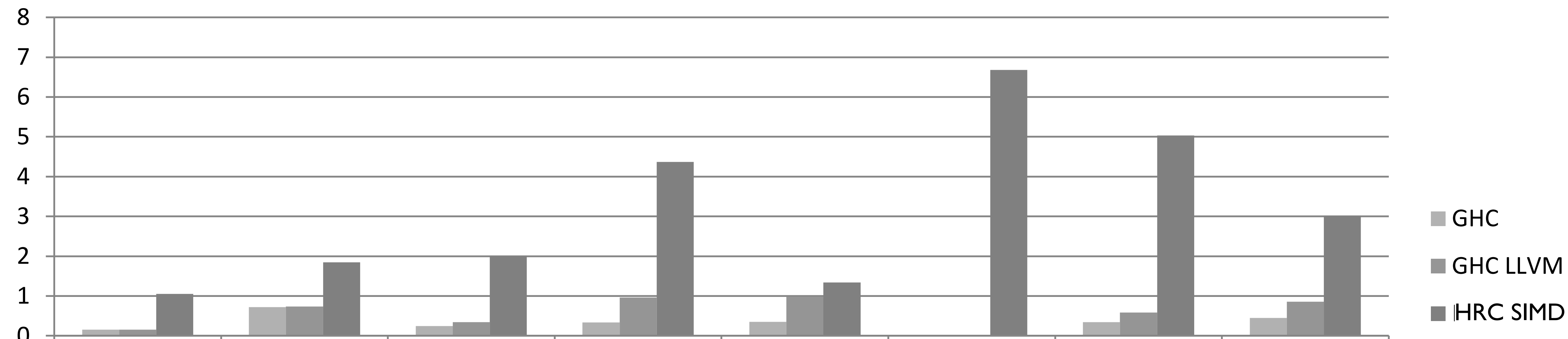
We told ‘em we could do parallelism!

Backup Slides

Pillar

- C-like language
- Managed memory with garbage-collection
- Tail calls
- Continuations
- Compiles to C

[1] Anderson, Glew, Guo, Lewis, Liu, Liu, Petersen, Rajagopalan, Stichnoth, Wu, and Zhang. Pillar: A parallel implementation language. Languages and Compilers for Parallel Computing, Springer-Verlag, 2008



	Vector Add	Vector Sum	Dot Product	N Body Repa	Matrix Multiply	1D Convolution	2D Convolution	Blur
GHC	0.15	0.72	0.24	0.34	0.35	0.02	0.34	0.45
GHC LLVM	0.15	0.73	0.35	0.97	0.98	0.02	0.58	0.86
HRC SIMD	1.05	1.84	2.00	4.37	1.34	6.68	5.03	3.02