# Data Parallelism in Haskell

**Manuel M. T. Chakravarty**
University of New South Wales

Includes joint work with

Gabriele Keller
Sean Lee
Roman Leshchinskiy
Simon Peyton Jones

# My three main points

1. Parallel programming and functional programming are intimately connected

2. Data parallelism is cheaper than control parallelism

3. Two approaches to data parallelism in Haskell

PLS

# Parallel ⟷ Functional

- What is hard about parallel programming?
- Why is it easier in a functional language?

# What is Hard About Parallelism?

# What is Hard About Parallelism?

- Indeterminate execution order!
- Other difficulties are arguably a consequence (race conditions, mutual exclusion, and so on)

# Why Use a Functional Language?

# Why Use a Functional Language?

- De-emphasises attention to execution order
  - Purity and persistance
  - Focus on data dependencies
- Encourages the use of collective operations
  - Wholemeal programming is better for you!

# Why Use a Functional Language?

- De-emphasises attention to execution order
  - Purity and persistance
  - Focus on data dependencies
- Encourages the use of collective operations
  - Wholemeal programming is better for parallelism!

# Haskell?

PLS

# Haskell?

- Laziness prevented bad habits
- Haskell programmers are not spoiled by the luxury of predictable execution order — a luxury that we can no longer afford in the presence of parallelism.
- Haskell programming culture and implementations avoid relying on a specific execution order

# Haskell?

- Lazines
- Haskell the luxury of predi that we can no arallelism.
- Haskell entations avoid relying on a specific execution order

**Haskell is ready for parallelism!**

PLS

# Why should we care about data parallelism?

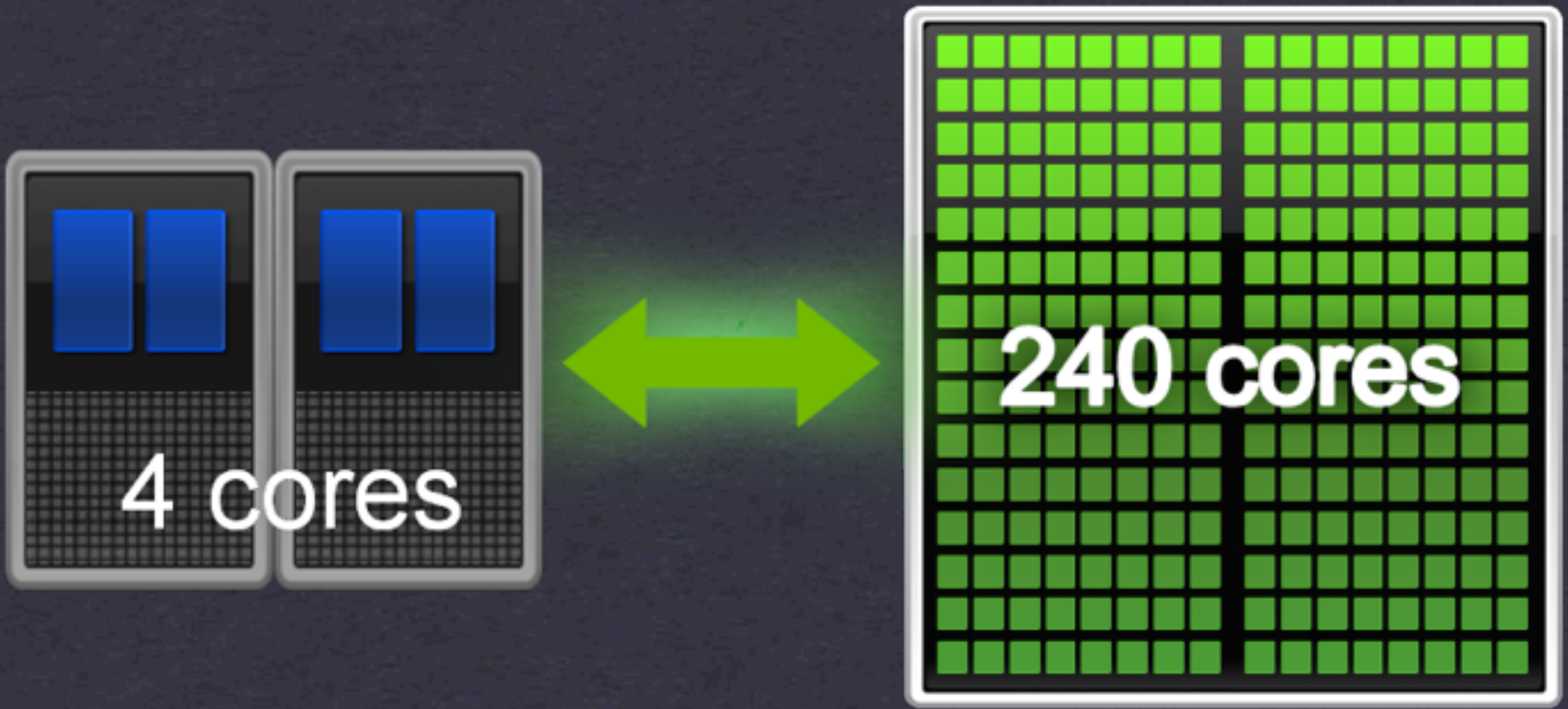
# Data parallelism is successful in the large

- On servers farms: CGI rendering, MapReduce, ...
- Fortran and OpenMP for high-performance computing

# Data parallelism is successful in the large

- On servers farms: CGI rendering, MapReduce, ...
- Fortran and OpenMP for high-performance computing

**Data parallelism becomes increasingly important in the small!**



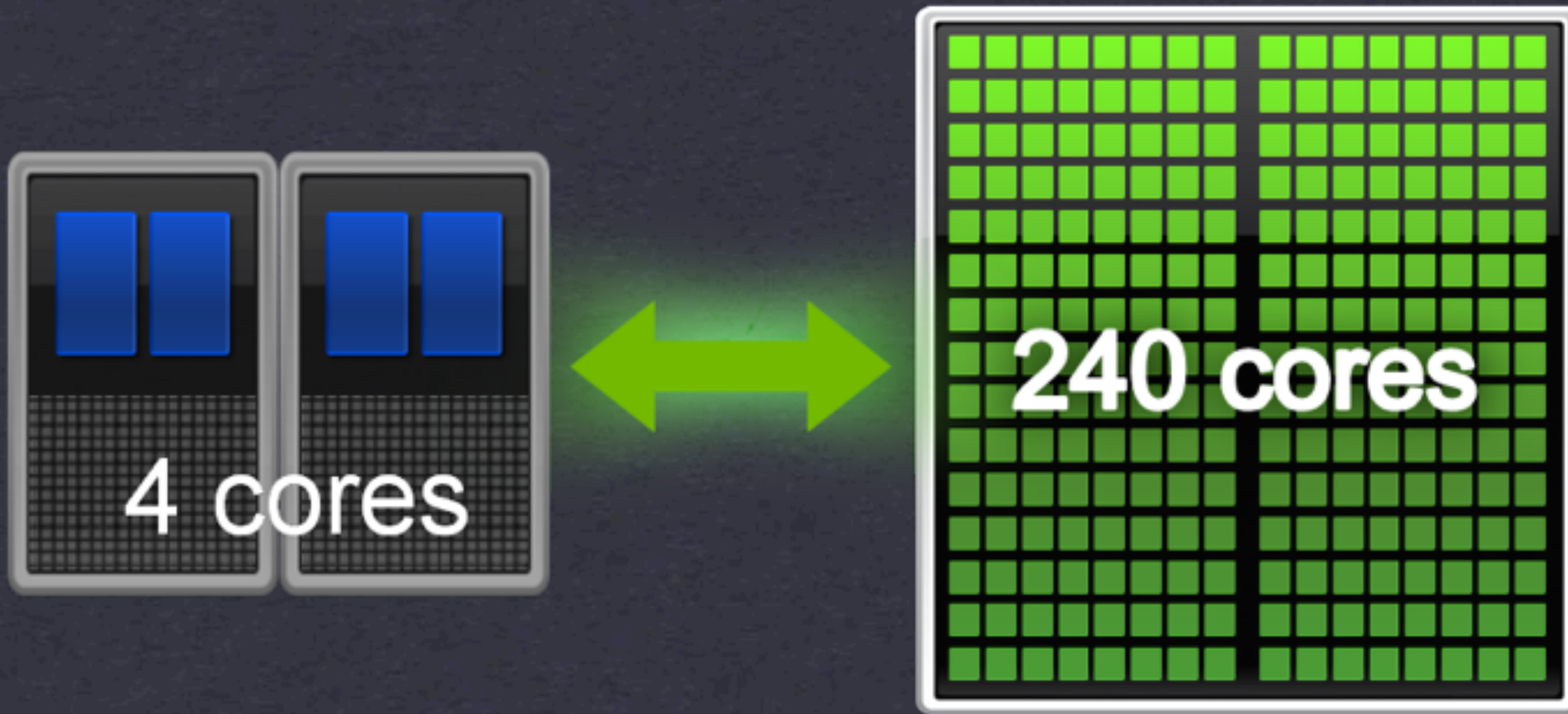

[Image courtesy of NVIDIA]

Quadcore
Xeon CPU

Tesla T10
GPU

# Our Data Parallel Future

Two competing extremes in current processor design




[Image courtesy of NVIDIA]

**Quadcore Xeon CPU**

**Tesla T10 GPU**

# Why?

## Our Data Parallel Future

Two competing extremes in current processor design

# Reduce power consumption!

- ✳ GPU achieves 20x better performance/Watt (judging by peak performance)
- ✳ Speedups between 20x to 150x have been observed in real applications

# We need data parallelism

- GPU-like architectures require data parallelism
- 4 core CPU versus 240 core GPU are the current extreme
- Intel Larrabee (in 2010): 32 cores x 16 vector units
- Increasing core counts in CPUs and GPUs

# We need data parallelism

- GPU-like architectures require data parallelism
- 4 core CPU versus 240 core GPU are the current extreme
- Intel Larrabee (in 2010): 32 cores x 16 vector units
- Increasing core counts in CPUs and GPUs

**Data parallelism is good news for functional programming!**

PLS

# Data parallelism and functional programming

**CUDA Kernel Invocation**

```
seq_kernel<<N, M>>(arg1, ..., argn);
```

# Data parallelism and functional programming

**CUDA Kernel Invocation**

```
seq_kernel<<N, M>>(arg1, ..., argn);
```

**FORTRAN 95**

```
FORALL (i=1:n)
  A(i,i) = pure_function(b,i)
END FORALL
```

PLS

# Data parallelism and functional programming

**CUDA Kernel Invocation**

```
seq_kernel<<N, M>>(arg1, ..., argn);
```

**FORTRAN 95**

```
FORALL (i=1:n)
  A(i,i) = pure_function(b,i)
END FORALL
```

- Parallel map is essential; reductions are common
- Parallel code must be pure

PLS

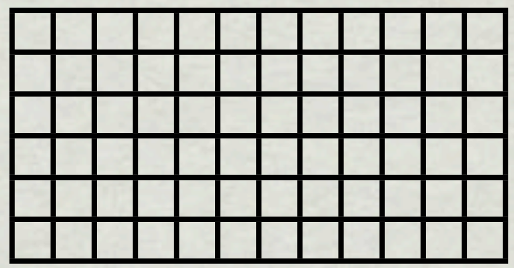
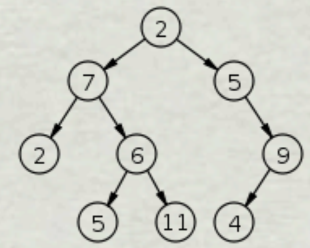
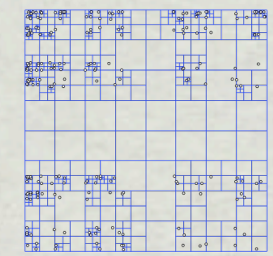
# Two Approaches to Data Parallel Programming in Haskell

PLS

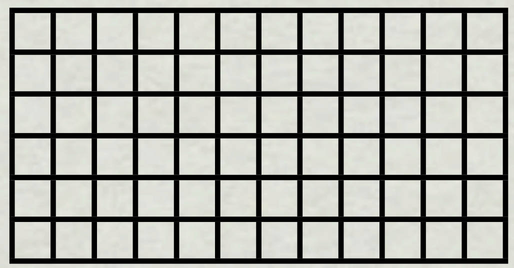
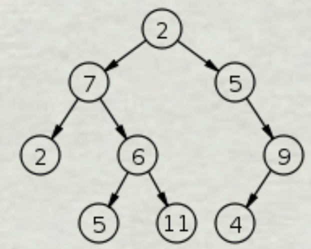
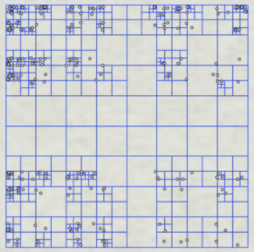
# Two forms of data parallelism

| flat, regular | nested, irregular |
|---|---|
| | |

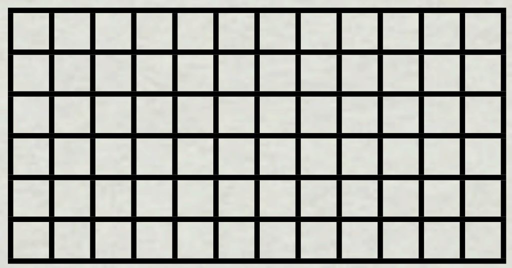
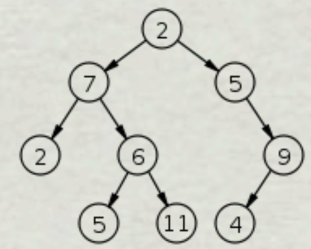
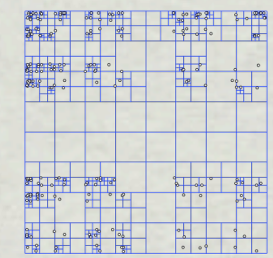
# Two forms of data parallelism

| flat, regular | | nested, irregular | |
|---|---|---|---|
|  |  |  |  |

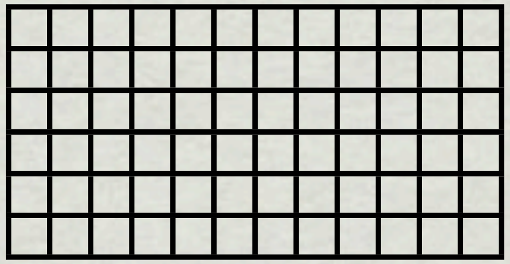
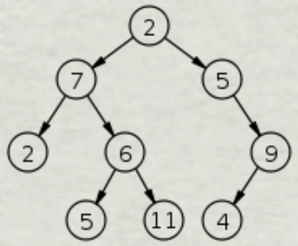
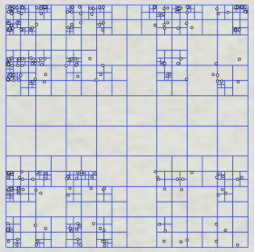
# Two forms of data parallelism

| flat, regular | nested, irregular |
| --- | --- |
|   |    |
| limited expressiveness | covers sparse structures and even divide&conquer |

# Two forms of data parallelism

| flat, regular | nested, irregular |
|---|---|
|   |    |
| limited expressiveness | covers sparse structures and even divide&conquer |
| close to the hardware model | needs to be turned into flat parallelism for execution |

# Two forms of data parallelism

| flat, regular | nested, irregular |
| --- | --- |
|   |    |
| limited expressiveness | covers sparse structures and even divide&conquer |
| close to the hardware model | needs to be turned into flat parallelism for execution |
| well understood compilation techniques | highly experimental program transformations |

# Flat data parallelism in Haskell

- Embedded language of array computations (two-level language)
- Datatype of multi-dimensional arrays [Gabi's talk]
- Array elements limited to tuples of scalars (`Int`, `Float`, `Bool`, etc)
- Collective array operations: map, fold, scan, zip, permute, etc.

PLS

# Scalar Alpha X Plus Y (SAXPY)

```
type Vector = Array DIM1 Float

saxpy :: GPU.Exp Float -> Vector -> Vector
      -> Vector
saxpy alpha xs ys
  = GPU.run $ do
      xs' <- use xs
      ys' <- use ys
      GPU.zipWith (\x y -> alpha*x + y) xs' ys'
```

PLS

# Scalar Alpha X Plus Y (SAXPY)

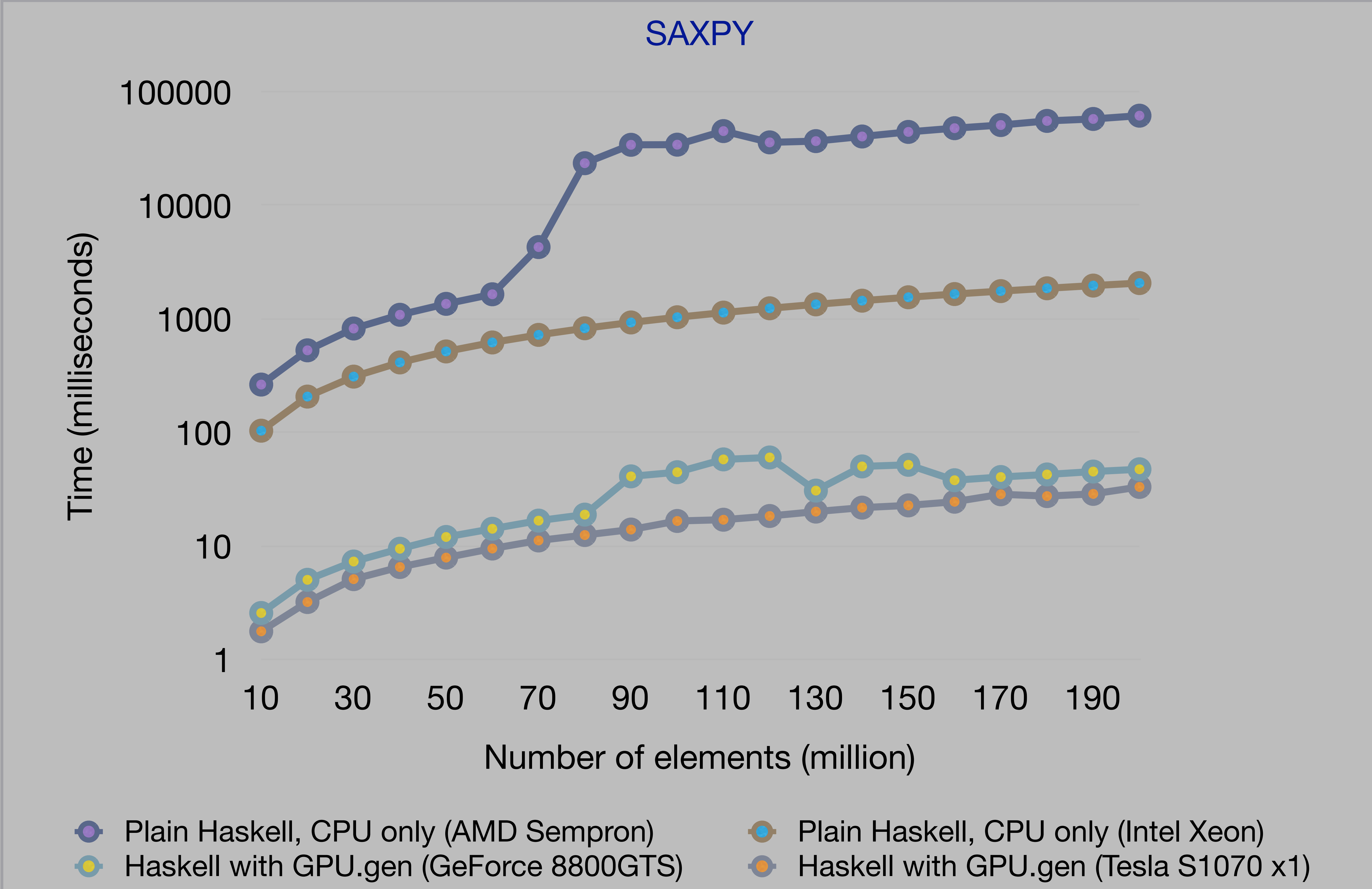
```
type Vector = Array DIM1 Float

saxpy :: GPU.Exp Float -> Vector -> Vector
      -> Vector
saxpy alpha xs ys
  = GPU.run $ do
      xs' <- use xs
      ys' <- use ys
      GPU.zipWith (\x y -> alpha*x + y) xs' ys'
```

- `GPU.Exp e` — expression evaluated on the GPU
- Monadic code to make sharing explicit
- `GPU.run` — compile & execute embedded code

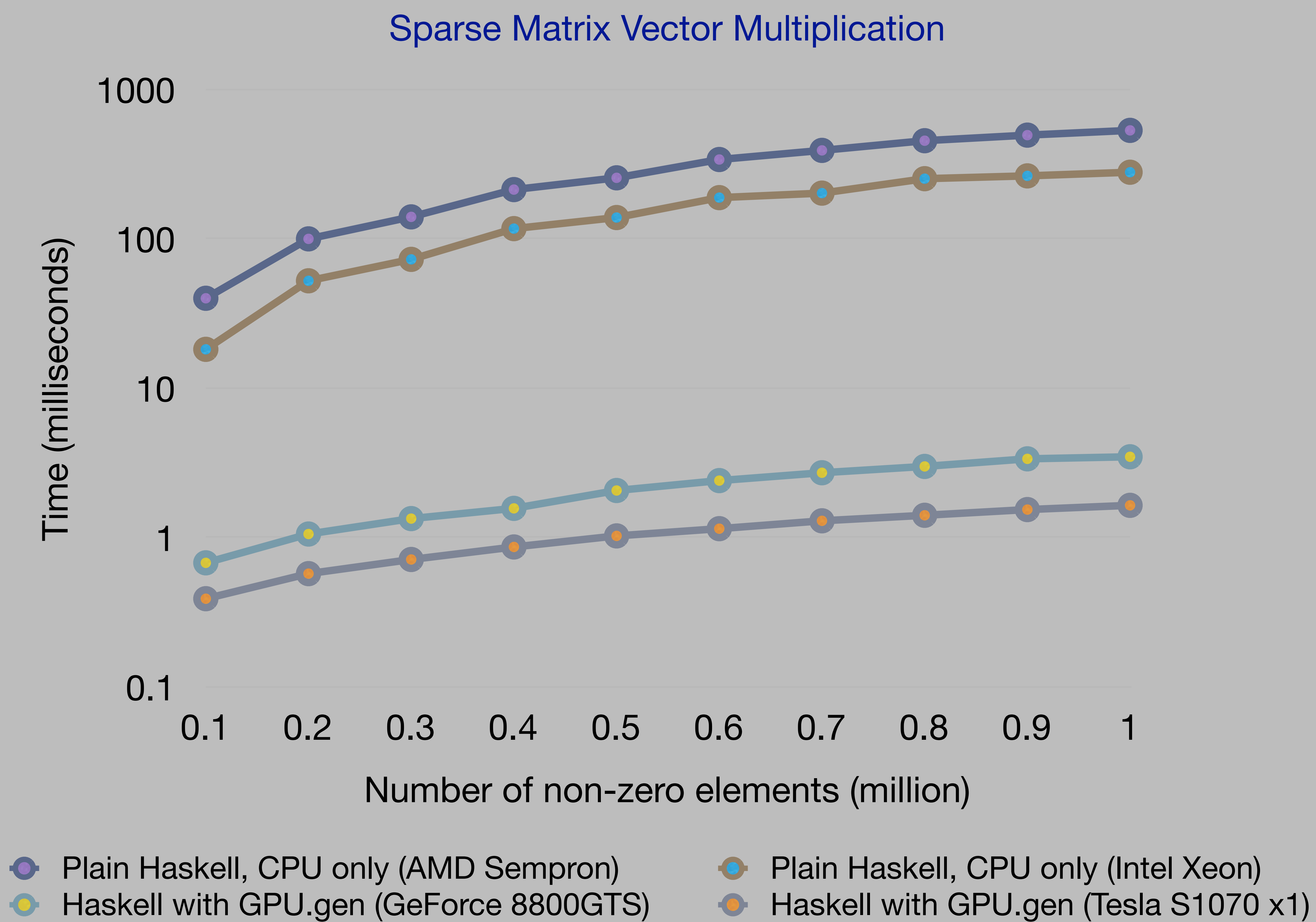
# Limitations of the embedded language

- First-order, except for a fixed set of higher-order collective operations
- No recursion
- No nesting — code is not compositional
- No arrays of structured data



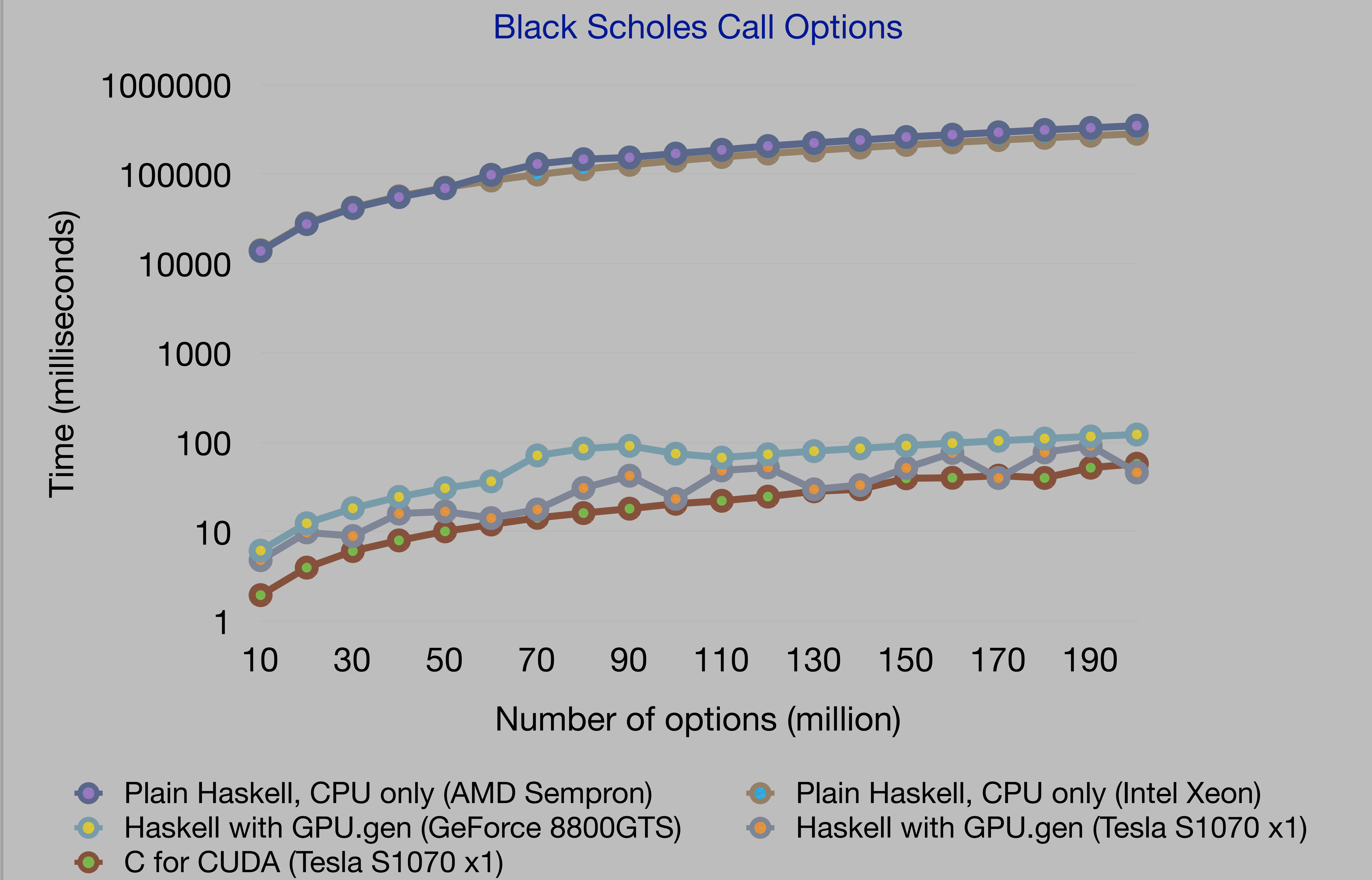

# Prototype implementation targeting GPUs

**Runtime code generation (computation only)**




# Prototype implementation targeting GPUs

**Runtime code generation (computation only)**




# Prototype implementation targeting GPUs

**Runtime code generation (computation only)**

# Nested data parallelism in Haskell

- Language extension (fully integrated)
- Data type of nested parallel arrays `[:e:]` — here, `e` can be any type
- Parallel evaluation semantics
- Array comprehensions & collective operations (`mapP`, `scanP`, etc.)
- *Forthcoming:* multidimensional arrays [Gabi's talk]

# Parallel Quicksort

```
qsort :: Ord a => [:a:] -> [:a:]
qsort [::] = [::]
qsort xs = let
             p       = xs!:0
             smaller = [:x | x <- xs, x < p:]
             equal   = [:x | x <- xs, x == p:]
             bigger  = [:x | x <- xs, x > p:]
             qs      = [:qsort xs‘
                        | xs‘ <- [:smaller, bigger:]:]
           in
           qs!:0 +:+ equal +:+ qs!:1
```

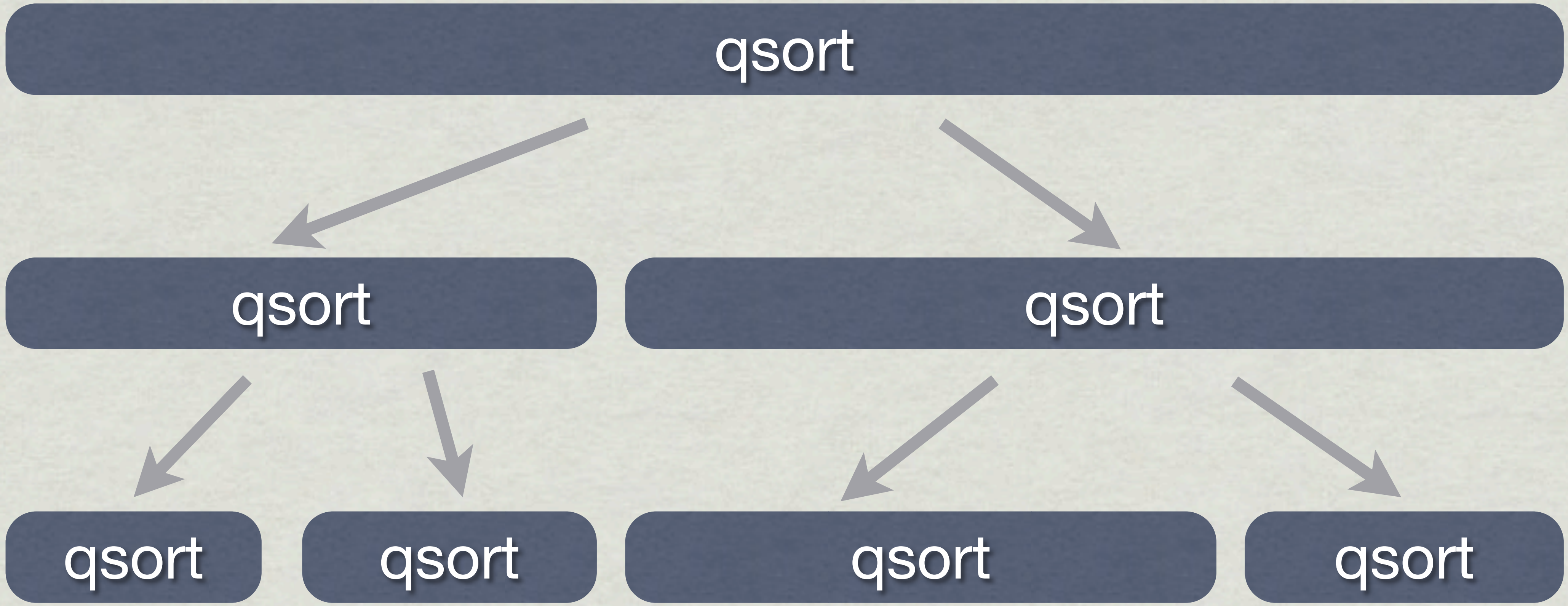
# Parallel Quicksort

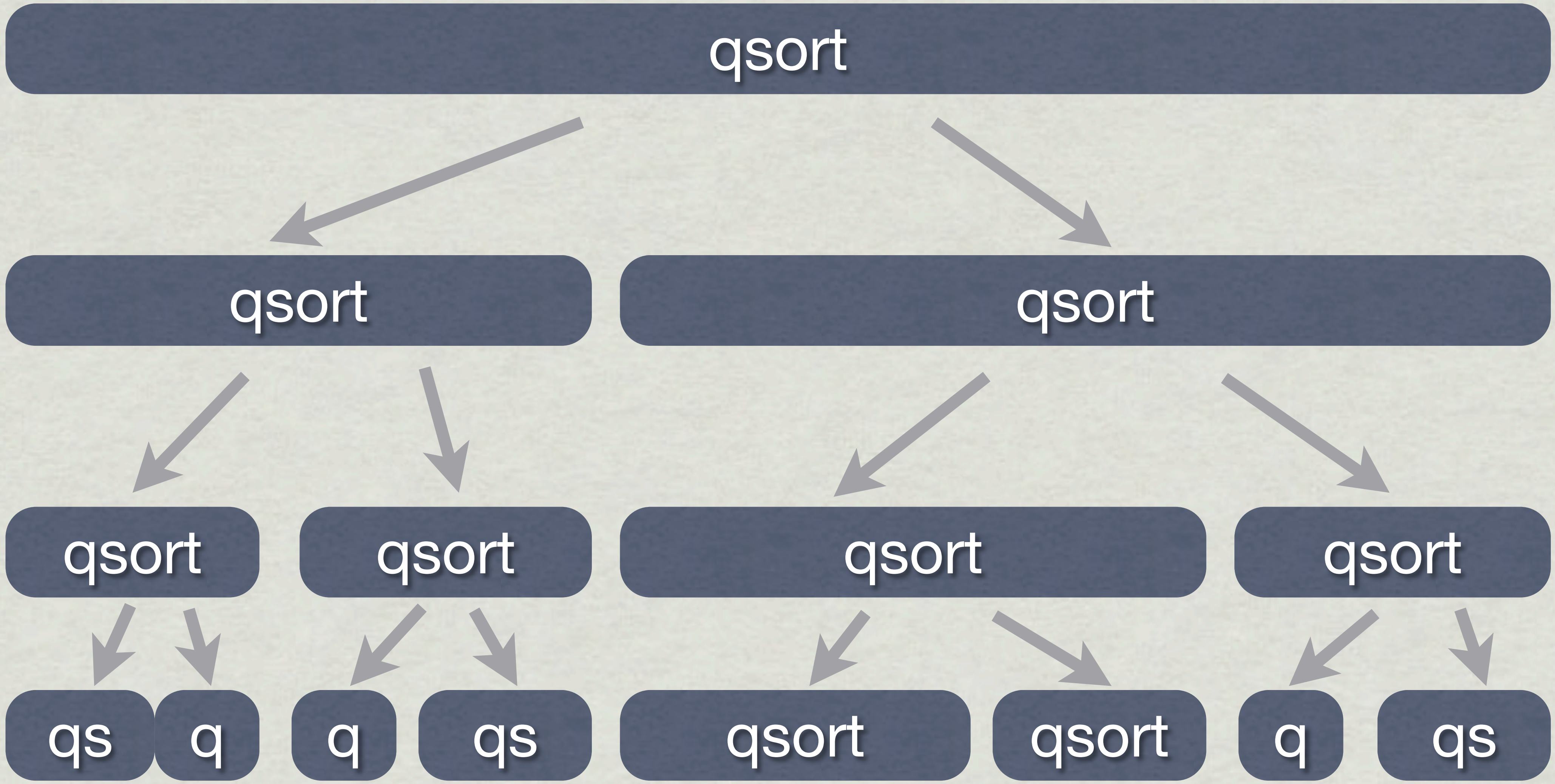
```
qsort :: Ord a => [:a:] -> [:a:]
qsort [::] = [::]
qsort xs = let
             p       = xs!:0
             smaller = [:x | x <- xs, x < p:]
             equal   = [:x | x <- xs, x == p:]
             bigger  = [:x | x <- xs, x > p:]
             qs      = [:qsort xs‘
                       | xs‘ <- [:smaller, bigger:]:]
           in
           qs!:0 +:+ equal +:+ qs!:1
```

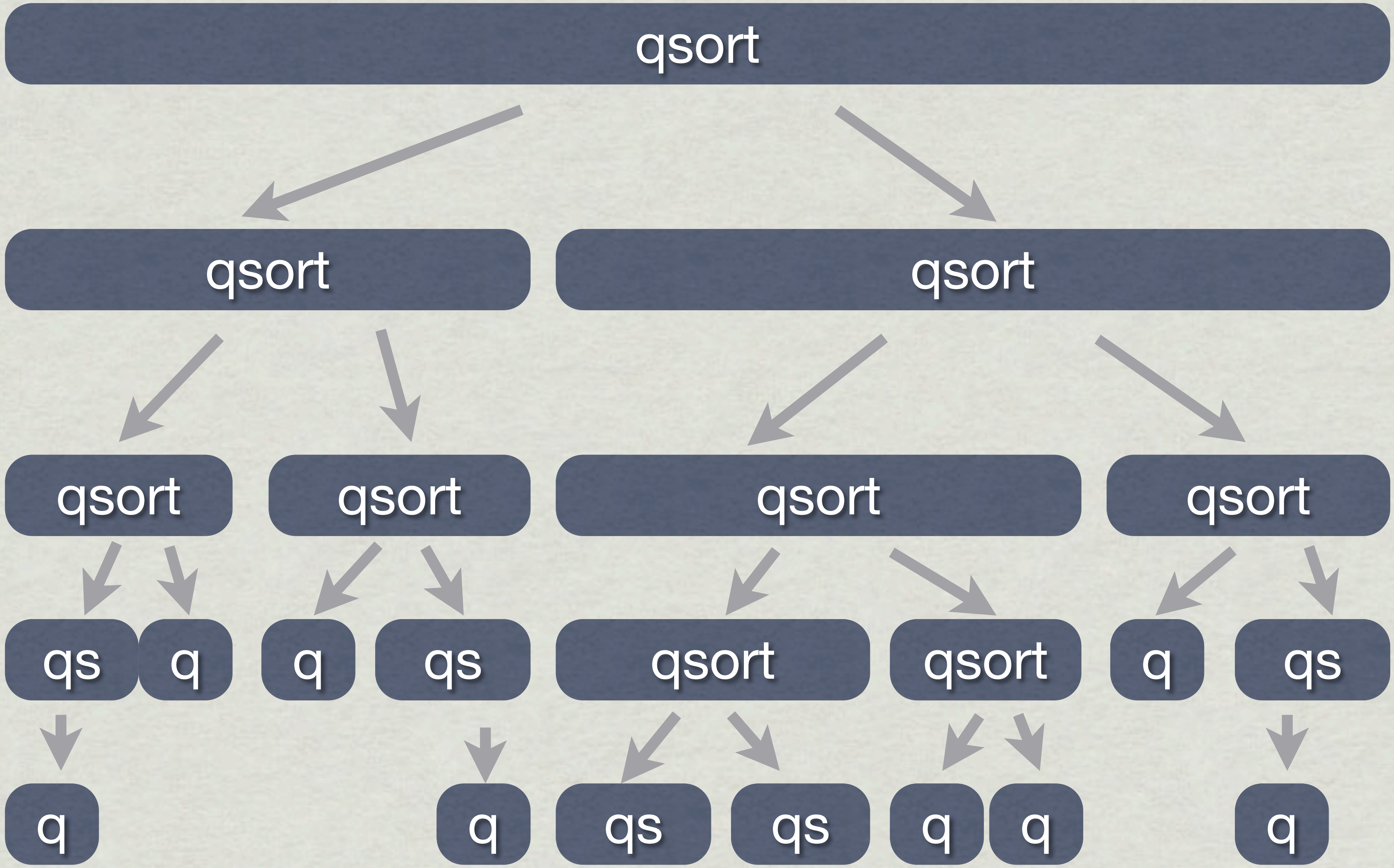
- `[: e | x <- xs:]` — array comprehension
- `(!:), (+:+)` — array indexing and append
- collective array operations are parallel

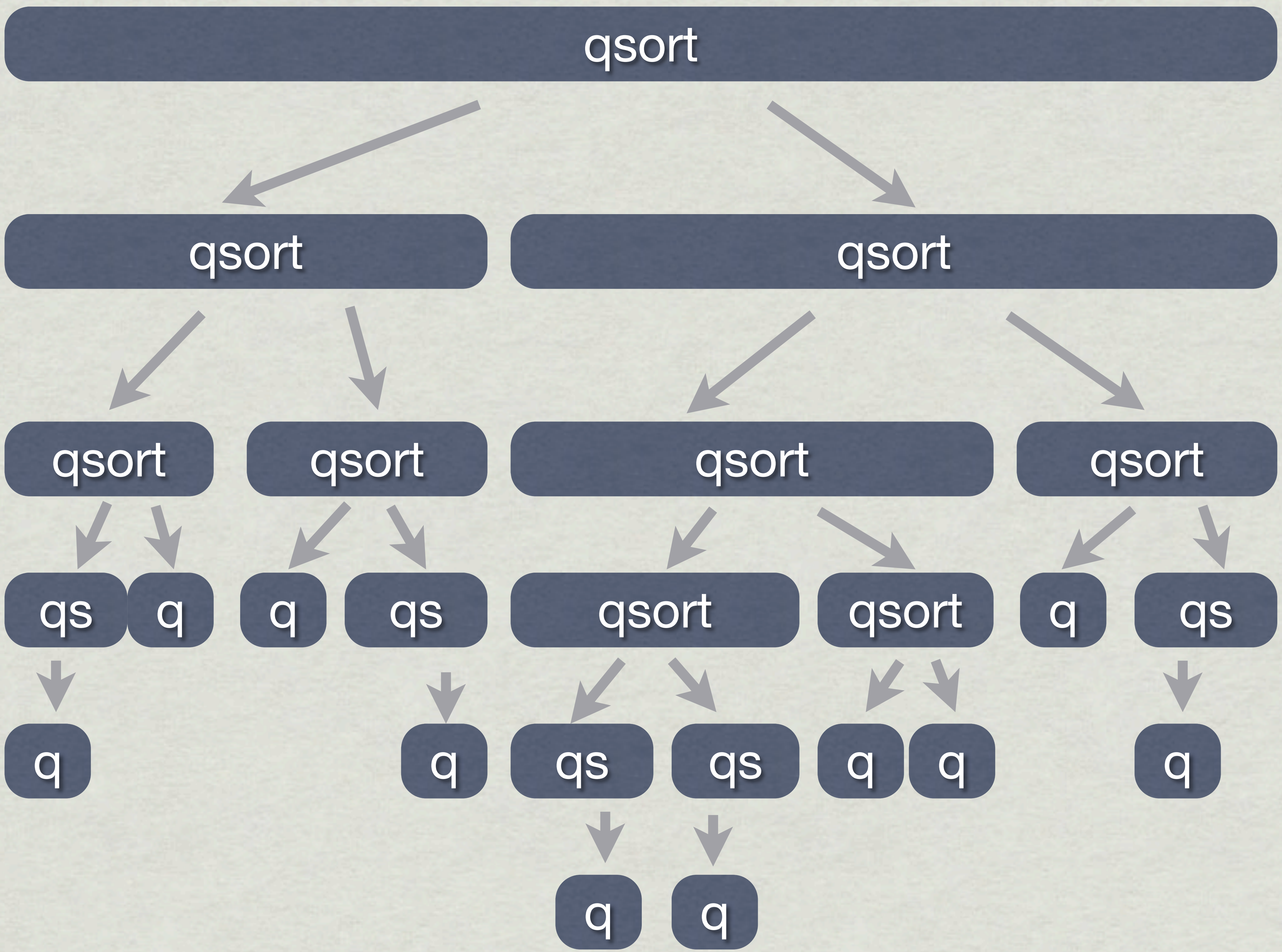
# qsort


qsort
qsort
qsort


qsort
qsort
qsort
qsort
qsort
qsort
qsort


qsort
qsort
qsort
qsort
qsort
qsort
qsort
qs
q
q
qs
qsort
qsort
q
qs


qsort
qsort
qsort
qsort
qsort
qsort
qsort
qs
q
q
qs
qsort
qsort
q
qs
q
q
qs
qs
q
q
q




**Exploiting both inner and intra function parallelism!**

# Properties of the language extension

- First class
- Arrays of structured data (e.g., arrays of trees)
  - `data RTree a = RTree a [:RTree a:]`
- Higher-order (e.g., parallel array of functions)
- Arbitrarily nested parallelism — compositional
- Much harder to implement!

PLS

# Implementation

* Extension of the Glasgow Haskell Compiler (GHC)

# Implementation

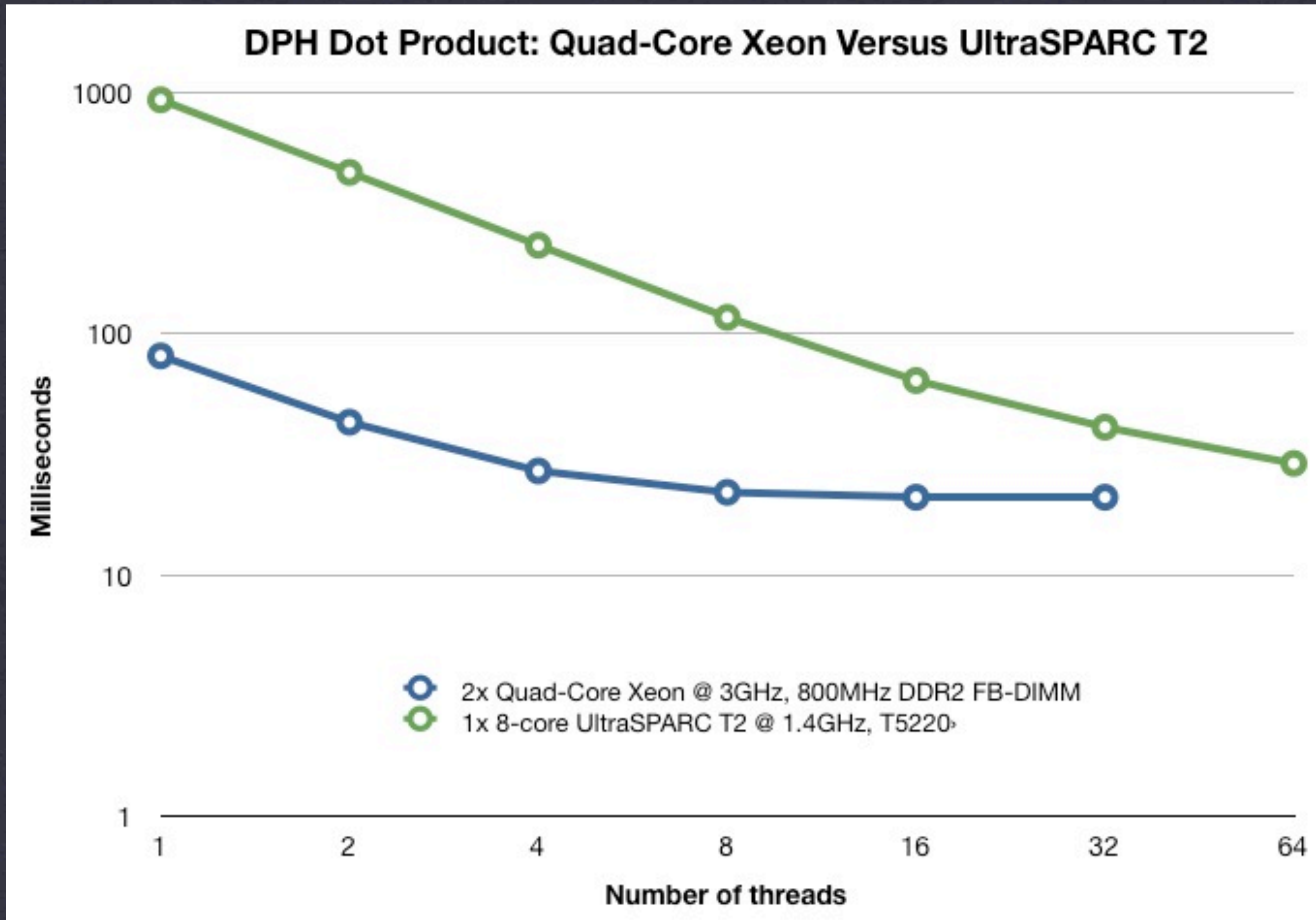
- Extension of the Glasgow Haskell Compiler (GHC)
- Stage 1: The Vectoriser
  - Transforms all nested into flat parallelism
  - `f :: a -> b`

# Implementation

- Extension of the Glasgow Haskell Compiler (GHC)
- Stage 1: The Vectoriser
    - Transforms all nested into flat parallelism
    - `f :: a -> b` → `f^ :: [:a:] -> [:b:]`

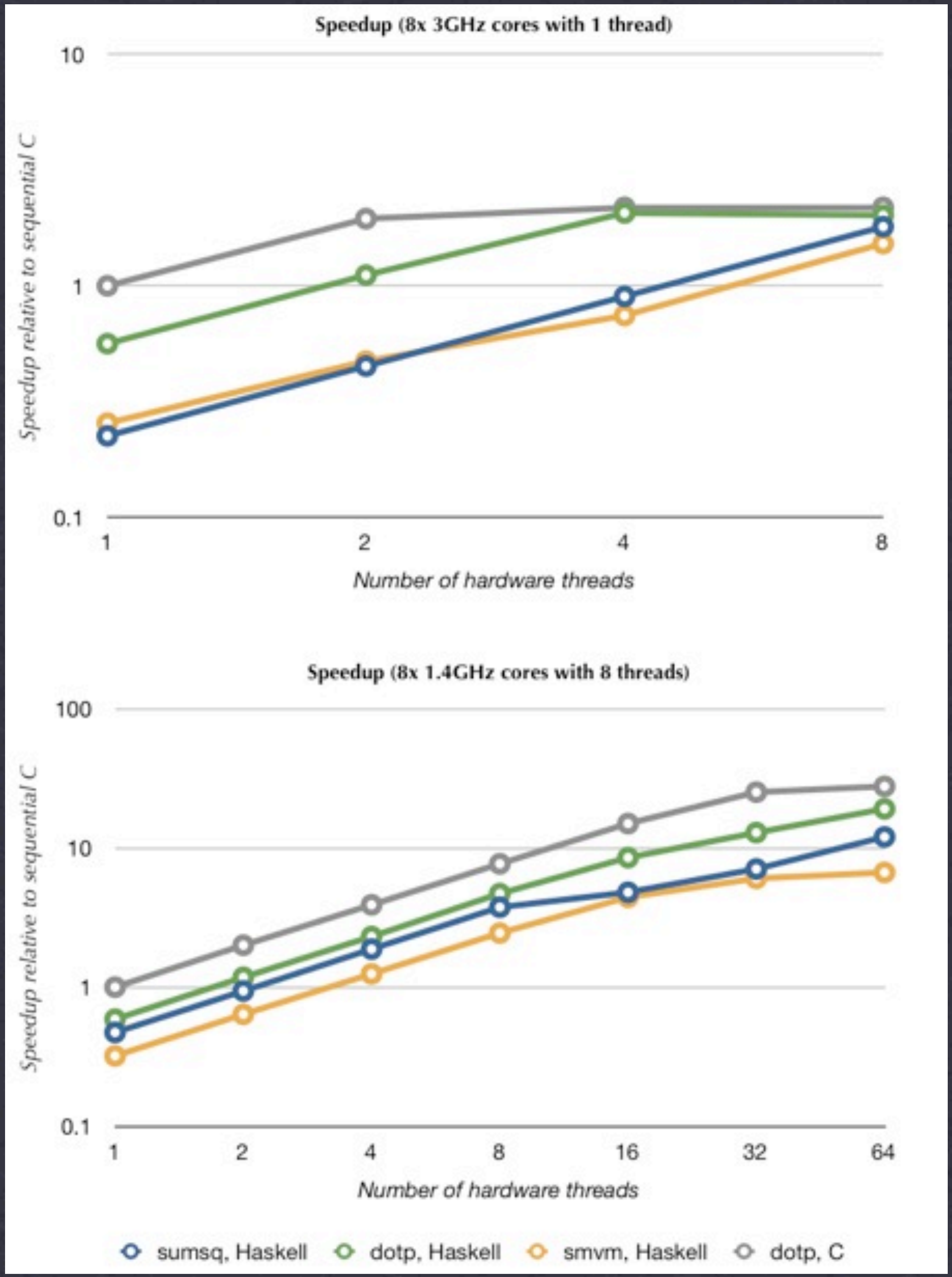
# Implementation

- Extension of the Glasgow Haskell Compiler (GHC)
- Stage 1: The Vectoriser
  - Transforms all nested into flat parallelism
  - `f :: a -> b` ➞ `f^ :: [:a:] -> [:b:]`
- Stage 2: Library package DPH
  - High-performance flat array library
  - Communication and array fusion
- Radical re-ordering of computations




# Current Implementation targeting multicore CPUs

## GHC performs vectorisation transformation on Core IL




2x Quad-Core Xeon = 8 cores
(8 thread contexts)

1x UltraSPARC T2 = 8 cores
(64 thread contexts)

# Current Implementation targeting multicore CPUs

GHC performs vectorisation transformation on Core IL

# Summary

- Data parallelism is getting increasingly important
- Two approaches to data parallelism in Haskell:
  1. Embedded array language for flat parallelism
  2. Language extension of parallel arrays supporting nested parallelism
- Nested parallelism is much harder to implement, but also much more expressive
- Multiple backends (multicore CPUs, GPUs, ...)