Data Parallel Programming in Haskell

An Overview

Manuel M T Chakravarty
University of New South Wales

Includes joint work with
Gabriele Keller
Sean Lee
Roman Leshchinskiy
Ben Lippmeier
Trevor McDonell
Simon Peyton Jones

Ubiquitous Parallelism

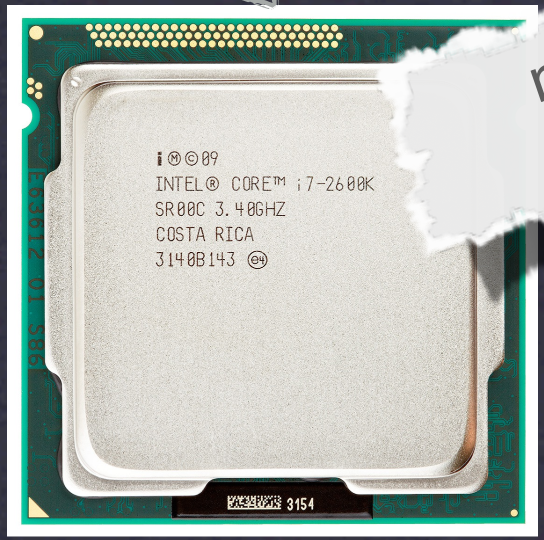
http://en.wikipedia.org/wiki/File:Cray_Y-MP_GSFC.jpg

venerable supercomputer

It's parallelism

http://en.wikipedia.org/wiki/File:Cray_Y-MP_GSFC.jpg

venerable supercomputer



multicore CPU

http://en.wikipedia.org/wiki/File:Intel_CPU_Core_i7_2600K_Sandy_Bridge_top.jpg

multicore GPU

http://en.wikipedia.org/wiki/File:GeForce_GT_545_DDR3.jpg

It's parallelism

...but not as we know it!

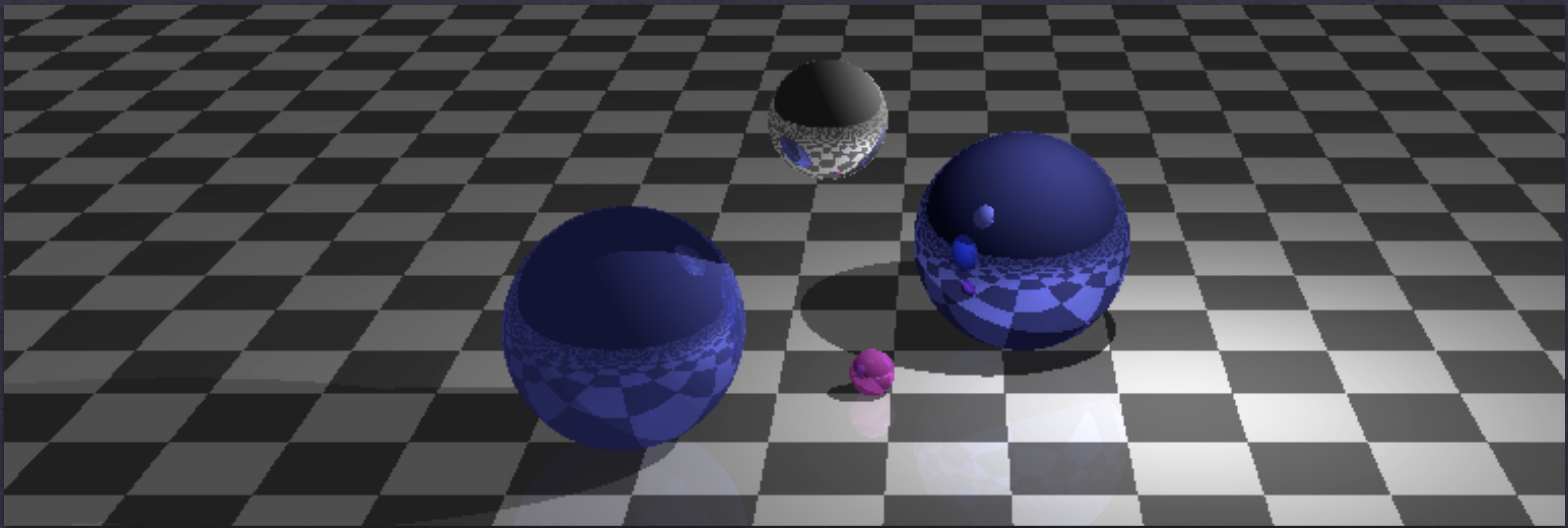
Our goals

Our goals

- Exploit parallelism of commodity hardware easily:
  - Performance is important, but…
  - …productivity is more important.

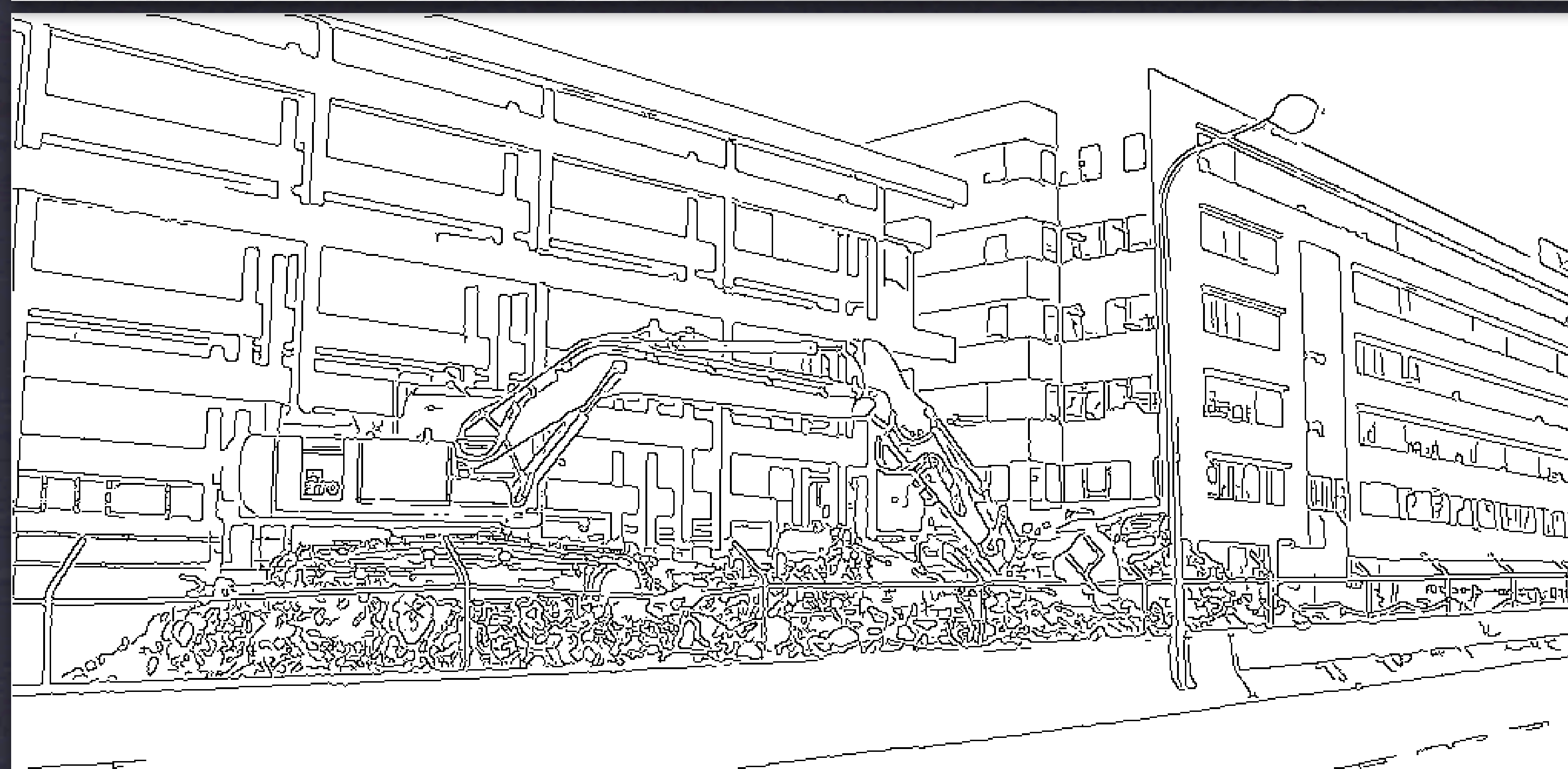
Our goals

- Exploit parallelism of commodity hardware easily:
  - Performance is important, but…
  - …productivity is more important.
- Semi-automatic parallelism
  - Programmer supplies a parallel algorithm
  - No explicit concurrency (no concurrency control, no races, no deadlocks)



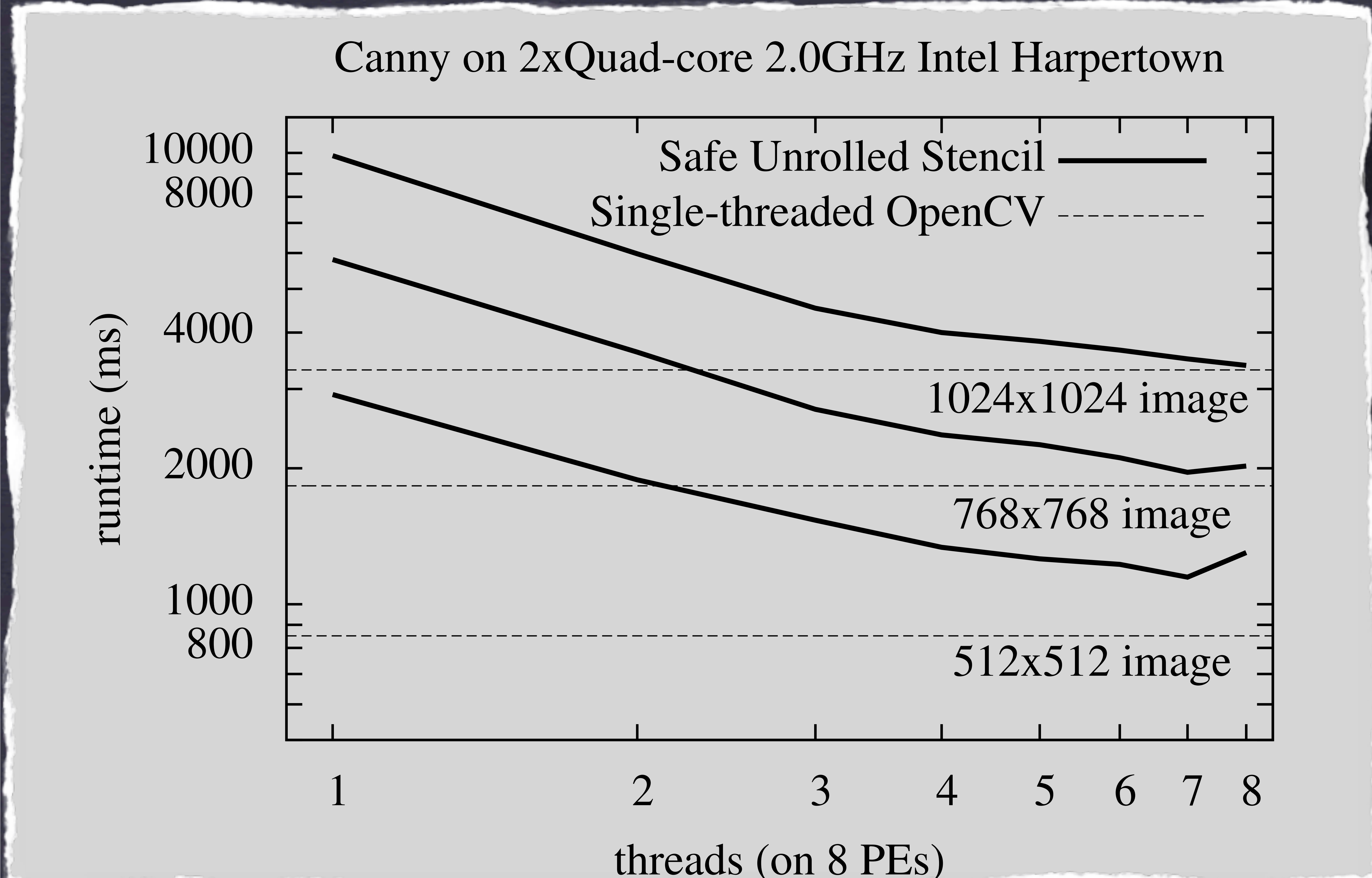
Graphics [Haskell 2012a]

Ray tracing



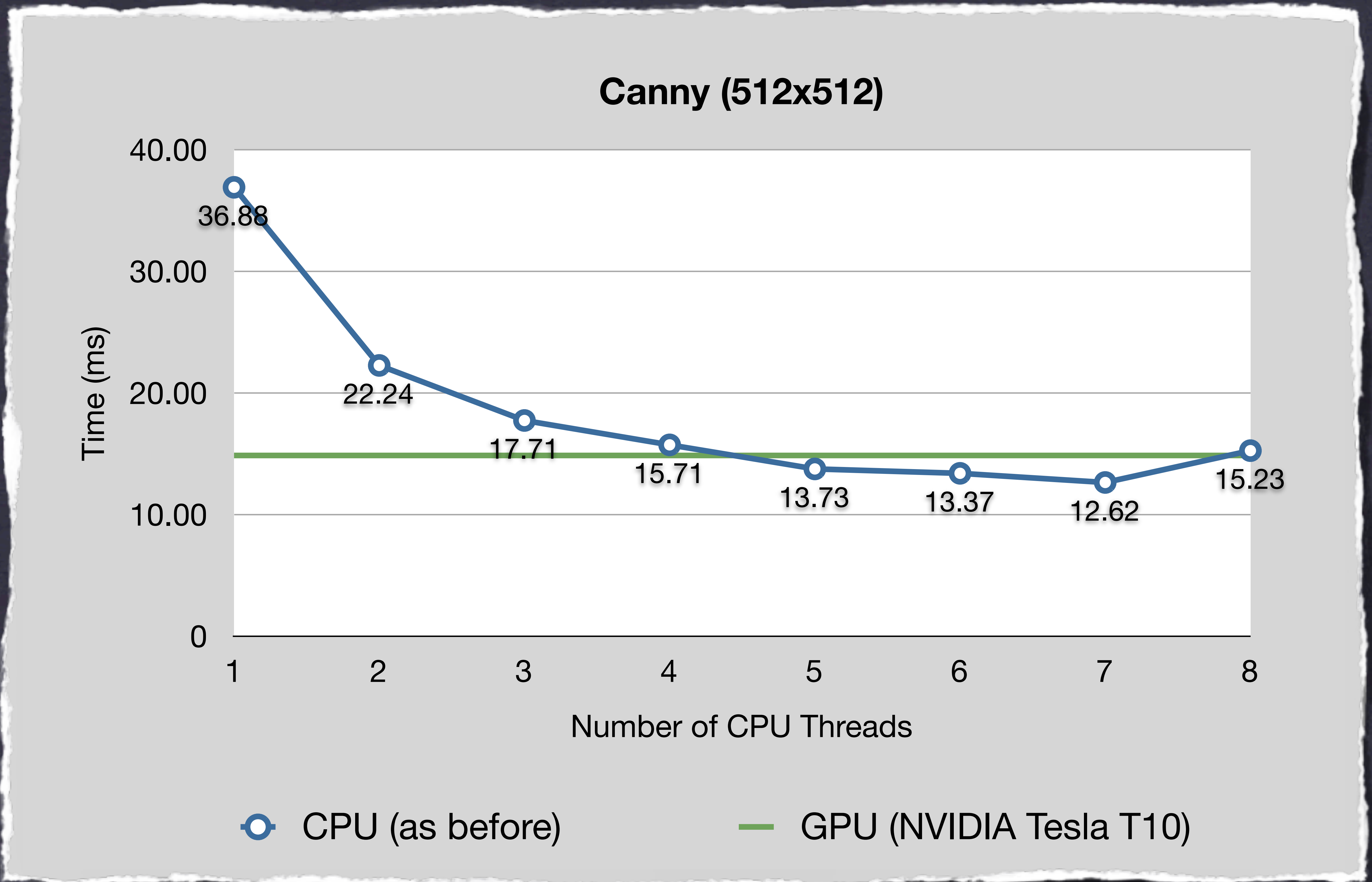
Computer Vision [Haskell 2011]

Edge detection



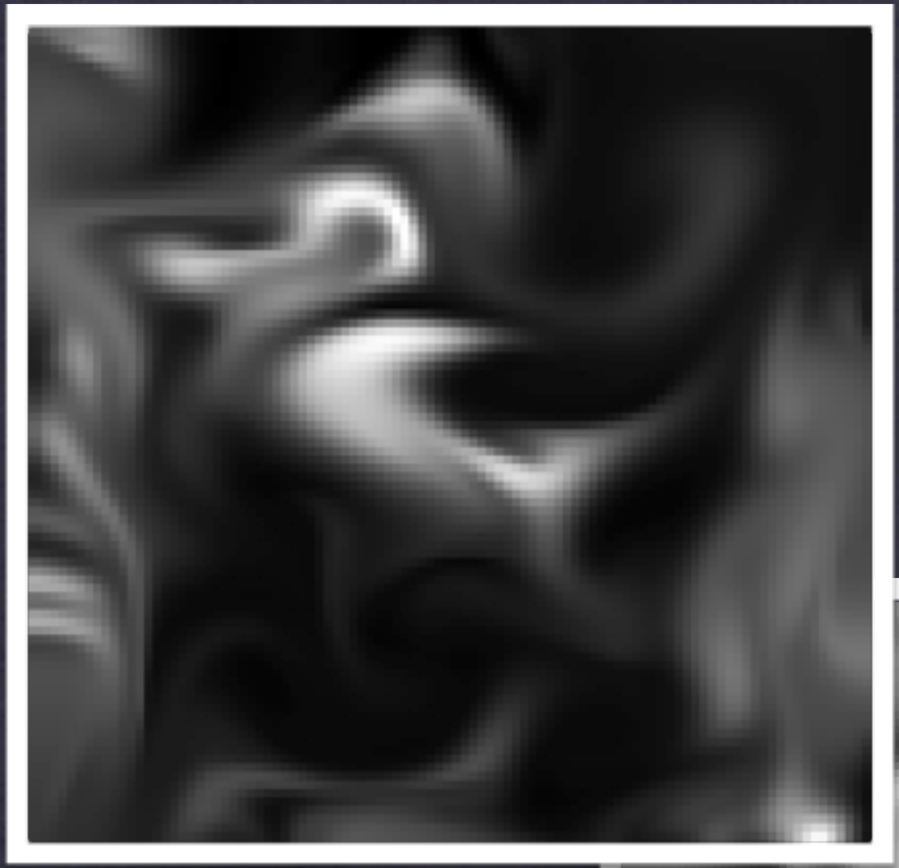
Runtimes for Canny edge detection (100 iterations)

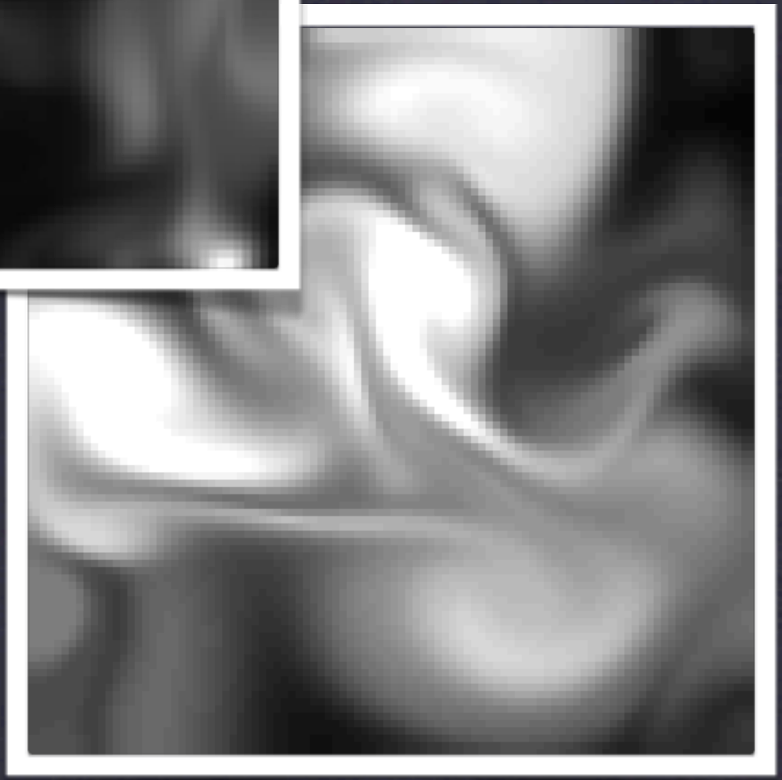
OpenCV uses SIMD-instructions, but only one thread



Canny edge detection: CPU versus GPU parallelism

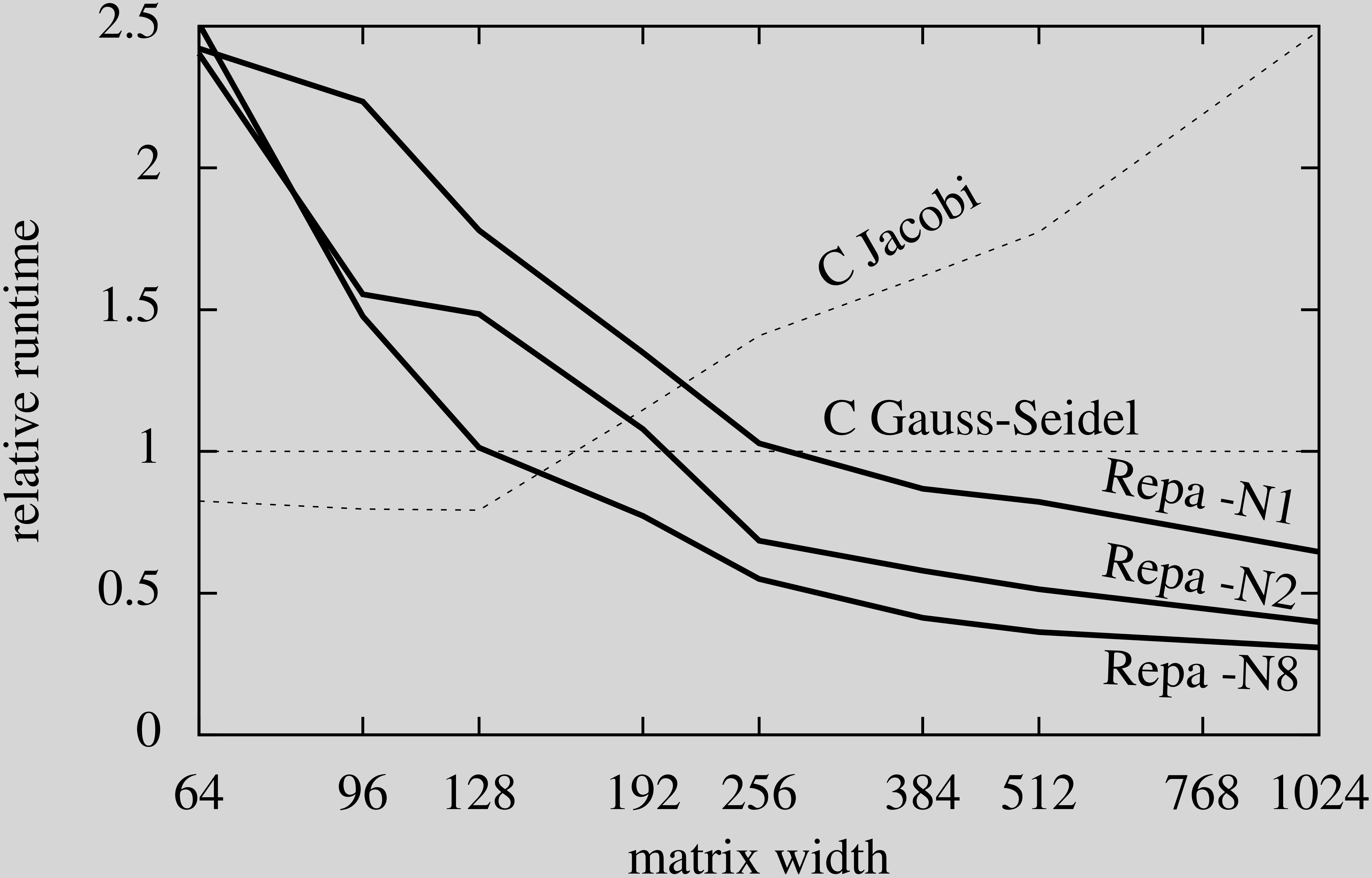
GPU version performs post-processing on the CPU





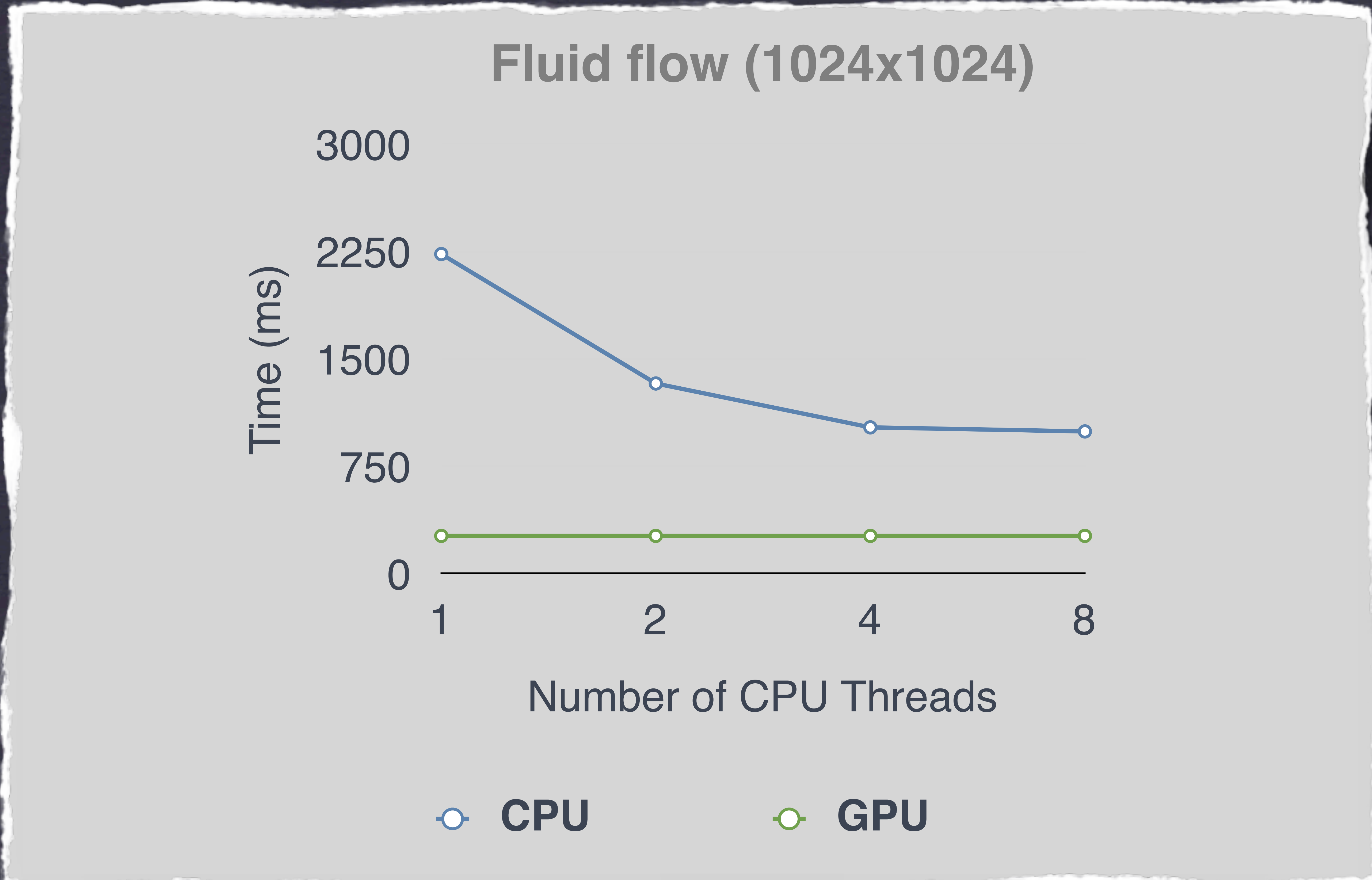
Physical Simulation [Haskell 2012a]

Fluid flow



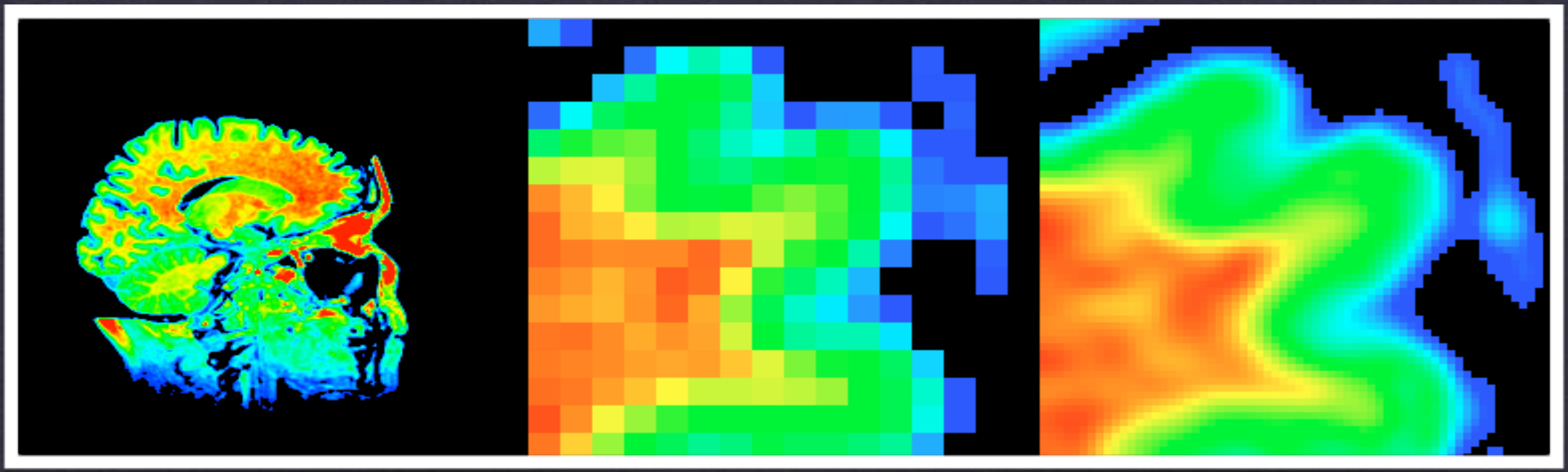
Runtimes for Jos Stam's Fluid Flow Solver

We can beat C!



Jos Stam's Fluid Flow Solver: CPU versus GPU Performance

GPU beats the CPU (includes all transfer times)



Medical Imaging [Haskell 2012a]

Interpolation of a slice though a 256 × 256 × 109 × 16-bit data volume of an MRI image

Functional Parallelism

Our ingredients

- Control effects, not concurrency
- Types guide data representation and behaviour
- Bulk-parallel aggregate operations

Our ingredients

Haskell is by default pure

- Control effects, not concurrency
- Types guide data representation and behaviour
- Bulk-parallel aggregate operations

PLS

Our ingredients

Haskell is by default pure

Declarative control of operational pragmatics

- Control effects, not concurrency
- Types guide data representation and behaviour
- Bulk-parallel aggregate operations

Our ingredients

- **Control effects**, not concurrency
- **Types** guide data representation and behaviour
- **Bulk-parallel** aggregate operations

Haskell is by default pure

Declarative control of operational pragmatics

Data parallelism

PLS

Purity & Types

Purity and parallelism

```
processList :: [Int] -> ([Int], Int)
processList list = (sort list, maximum list)
```

Purity and parallelism

```
processList :: [Int] -> ([Int], Int)
processList list = (sort list, maximum list)
```

function

argument

function body

PLS

Purity and parallelism

argument type

result type

```
processList :: [Int] -> ([Int], Int)
processList list = (sort list, maximum list)
```

function

argument

function body

Purity and parallelism

argument type · result type

```
processList :: [Int] -> ([Int], Int)
processList list = (sort list, maximum list)
```

function · argument · function body

- Purity: function result depends only on arguments

Purity and parallelism

argument type

result type

```
processList :: [Int] -> ([Int], Int)
processList list = (sort list, maximum list)
```

function

argument

function body

- **Purity:** function result depends only on arguments
- **Parallelism:** execution order only constrained by explicit data dependencies

PLS

By default pure

Types track purity

By default pure

Types track purity	
Pure = no effects	Impure = may have effects

By default pure

Types track purity	
Pure = no effects	Impure = may have effects
`Int`	`IO Int`

By default pure

Types track purity	
Pure = no effects	Impure = may have effects
`Int`	`IO Int`
`processList :: [Int] -> ([Int], Int)`	`readFile :: FilePath -> IO String`

By default pure

Types track purity	
Pure = no effects	Impure = may have effects
`Int`	`IO Int`
`processList ::` `  [Int] -> ([Int], Int)`	`readFile ::` `  FilePath -> IO String`

`(sort list, maximum list)`	`copyFile fn1 fn2 =`
	` do`
	`   data <- readFile fn1`
	`   writeFile fn2 data`

Types guide execution

Datatypes for parallelism

- For bulk-parallel, aggregate operations, we introduce a new datatype:

```
Array r sh e
```

Datatypes for parallelism

- For bulk-parallel, aggregate operations, we introduce a new datatype:

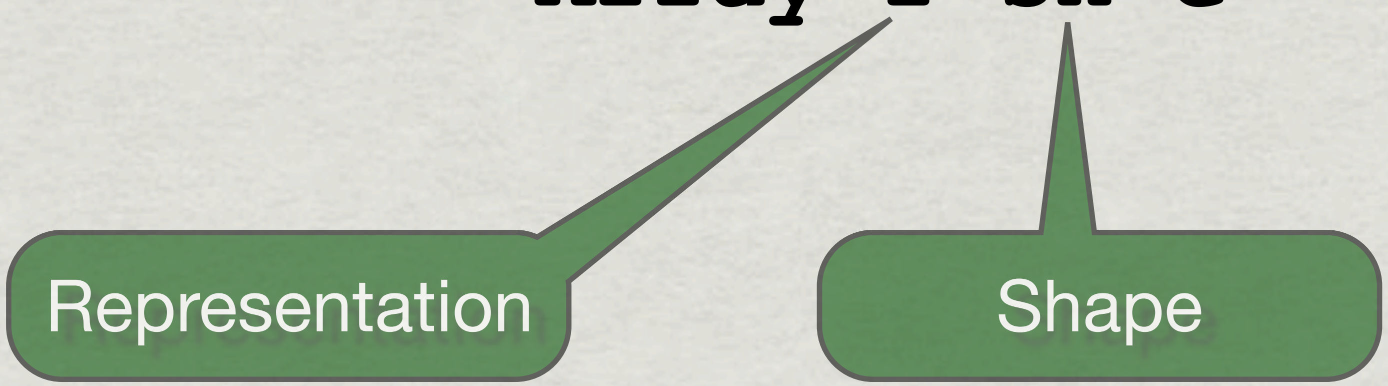
```
Array r sh e
```

Representation

Datatypes for parallelism

- For bulk-parallel, aggregate operations, we introduce a new datatype:

```
Array r sh e
```



Datatypes for parallelism

- For bulk-parallel, aggregate operations, we introduce a new datatype:

```
Array r sh e
```

Representation

Shape

Element type

Array r sh e

`Array r sh e`

- Representation: determined by a type index; e.g.,
  - `D` — delayed array (represented as a function)
  - `U` — unboxed array (manifest C-style array)

`Array r sh e`

- **Representation:** determined by a type index; e.g.,
  - `D` — delayed array (represented as a function)
  - `U` — unboxed array (manifest C-style array)
- **Shape:** dimensionality of the array
  - `DIM0`, `DIM1`, `DIM2`, and so on

`Array r sh e`

- Representation: determined by a type index; e.g.,
  - `D` — delayed array (represented as a function)
  - `U` — unboxed array (manifest C-style array)
- Shape: dimensionality of the array
  - `DIM0`, `DIM1`, `DIM2`, and so on
- Element type: stored in the array
  - Primitive types (`Int`, `Float`, etc.) and tuples

```haskell
zipWith :: (Shape sh, Source r1 a, Source r2 b)
        => (a -> b -> c)
        -> Array r1 sh a
        -> Array r2 sh b
        -> Array D sh c
```

```
zipWith :: (Shape sh, Source r1 a, Source r2 b)
        => (a -> b -> c)
        -> Array r1 sh a
        -> Array r2 sh b
        -> Array D sh c
```

Pure function to be used in parallel

```
zipWith :: (Shape sh, Source r1 a, Source r2 b)
        => (a -> b -> c)
        -> Array r1 sh a
        -> Array r2 sh b
        -> Array D sh c
```

Pure function to be used in parallel

Pocessed arrays

```haskell
zipWith :: (Shape sh, Source r1 a, Source r2 b)
        => (a -> b -> c)
        -> Array r1 sh a
        -> Array r2 sh b
        -> Array D sh c
```

Pure function to be used in parallel

Pocessed arrays

Delayed result

```
zipWith :: (Shape sh, Source r1 a, Source r2 b)
        => (a -> b -> c)
        -> Array r1 sh a
        -> Array r2 sh b
        -> Array D sh c
```

Pure function to be used in parallel

Delayed result

Pocessed arrays

```
type PC5
   = P C (P (S D)(P (S D)(P (S D)(P (S D) X))))
mapStencil2 :: Source r a
            => Boundary a
            -> Stencil DIM2 a
            -> Array r DIM2 a
            -> Array PC5 DIM2 a
```

```
zipWith :: (Shape sh, Source r1 a, Source r2 b)
        => (a -> b -> c)
        -> Array r1 sh a
        -> Array r2 sh b
        -> Array D sh c
```

Pure function to be used in parallel

Delayed result

Pocessed arrays

```
type PC5
   = P C (P (S D)(P (S D)(P (S D)(P (S D) X))))
mapStencil2 :: Source r a
            => Boundary a
            -> Stencil DIM2 a
            -> Array r DIM2 a
            -> Array PC5 DIM2 a
```

Partitioned result

A simple example — dot product

A simple example — dot product

```
dotp v w = sumAll (zipWith (*) v w)
```

A simple example — dot product

```
type Vector r e = Array r DIM1 e

dotp :: (Num e, Source r1 e, Source r2 e)
     => Vector r1 e -> Vector r2 e -> e
```

```
dotp v w = sumAll (zipWith (*) v w)
```

A simple example — dot product

Elements are any type of numbers…

```
type Vector r e = Array r DIM1 e

dotp :: (Num e, Source r1 e, Source r2 e)
     => Vector r1 e -> Vector r2 e -> e
```

```
dotp v w = sumAll (zipWith (*) v w)
```

A simple example — dot product

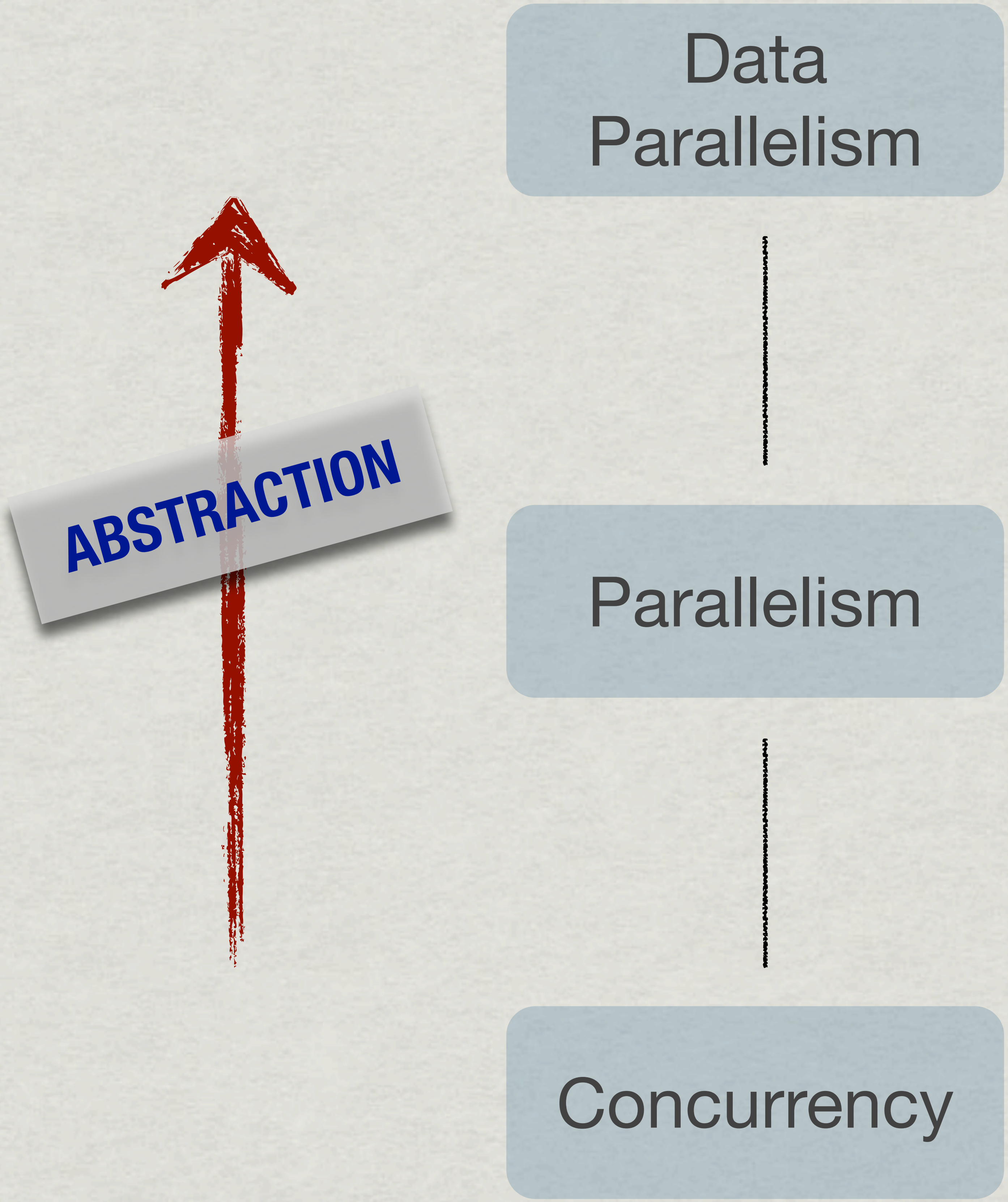
Elements are any type of numbers…

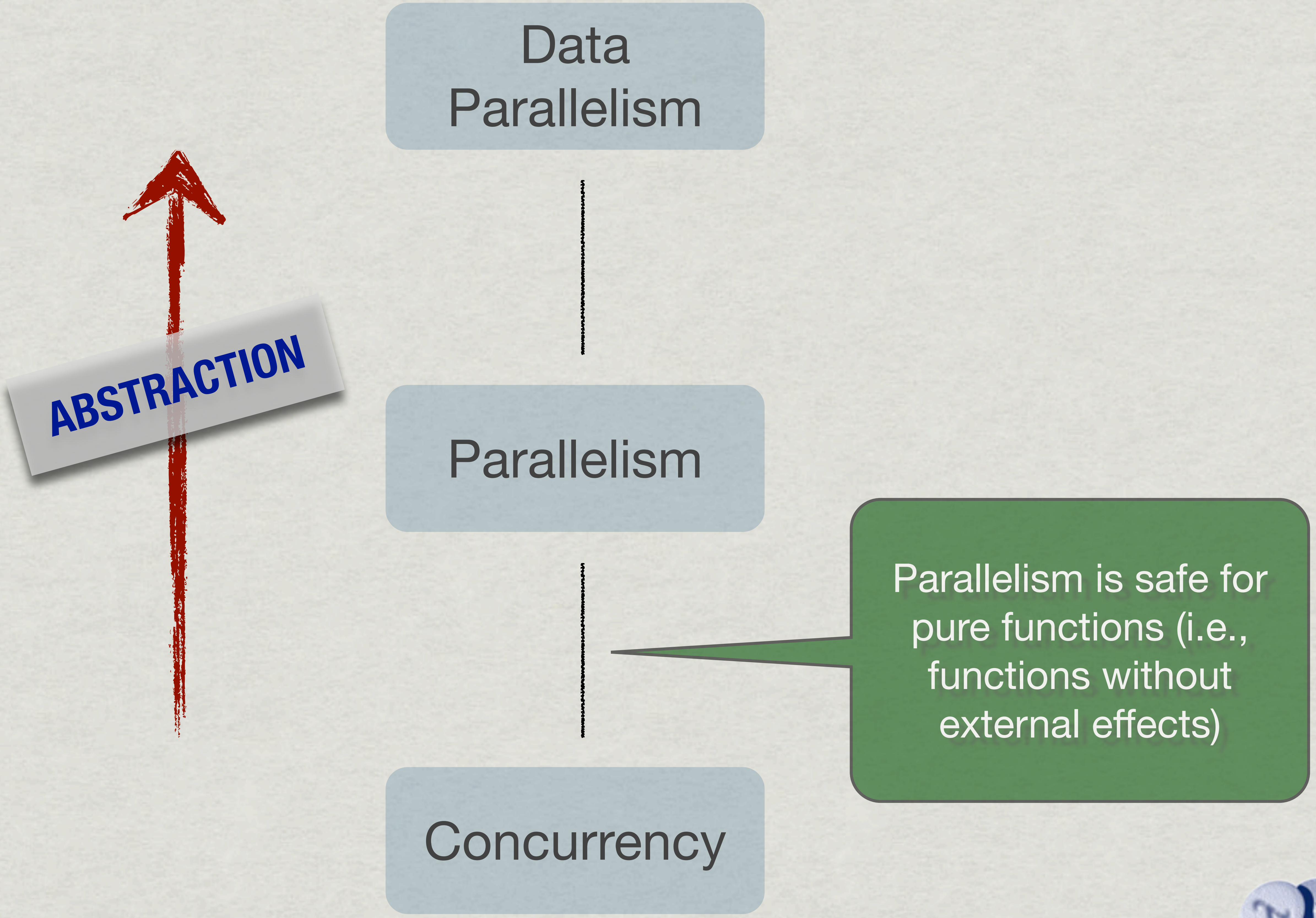
…suitable to be read from an array

```
type Vector r e = Array r DIM1 e

dotp :: (Num e, Source r1 e, Source r2 e)
     => Vector r1 e -> Vector r2 e -> e
```

```
dotp v w = sumAll (zipWith (*) v w)
```


Data Parallelism
Parallelism
Concurrency
ABSTRACTION


Data Parallelism
Parallelism
Concurrency
ABSTRACTION
Parallelism is safe for pure functions (i.e., functions without external effects)

PLS

ABSTRACTION

Data Parallelism

Collective operations have got a single conceptual thread of control

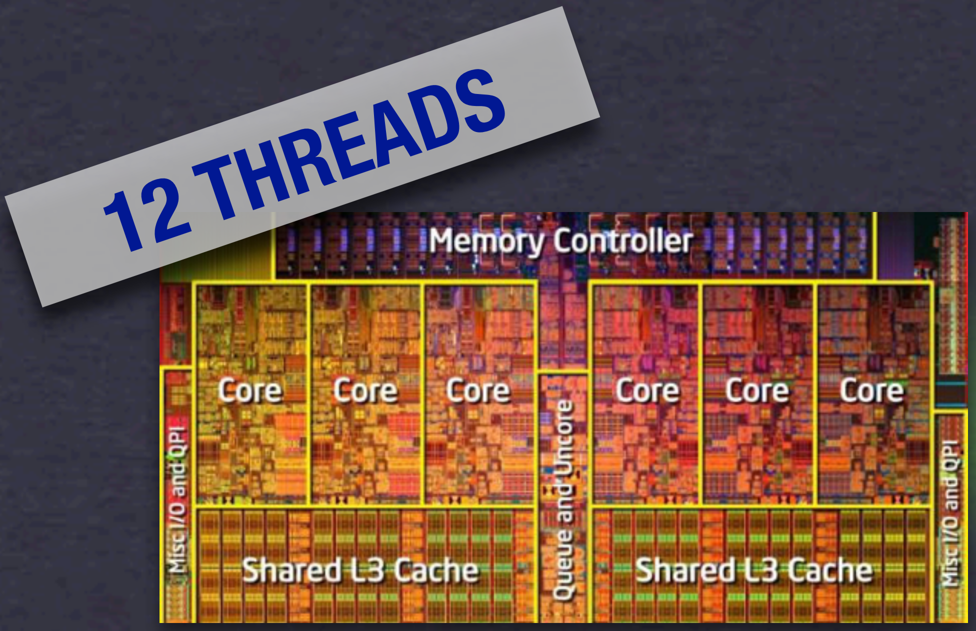
Parallelism

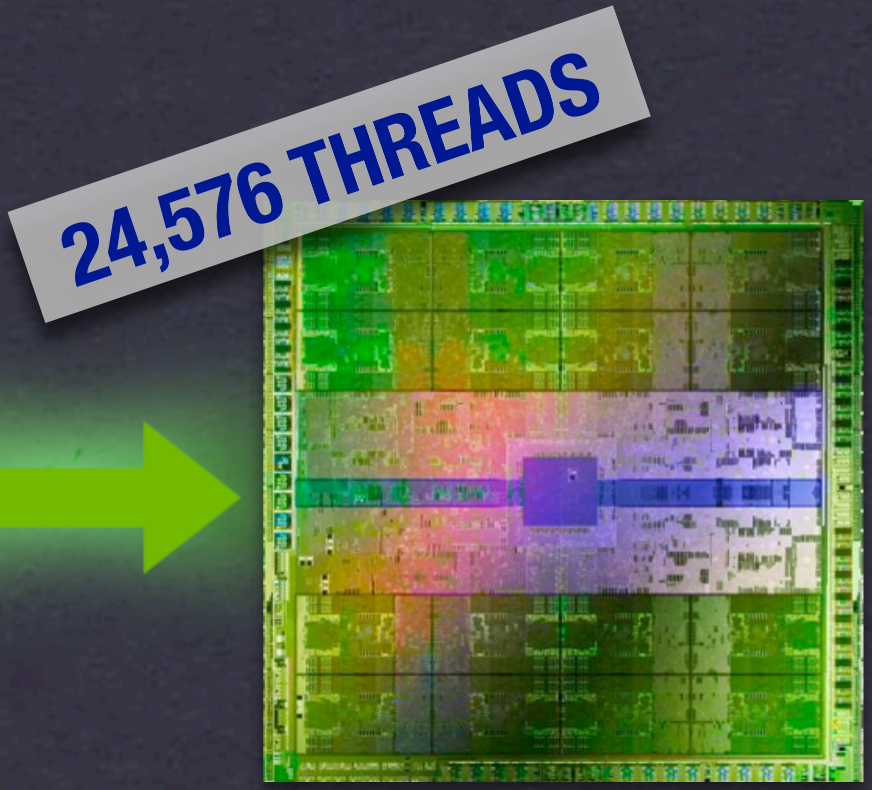
Parallelism is safe for pure functions (i.e., functions without external effects)

Concurrency

PLS

Types &
Embedded Computations





Core i7 970
CPU

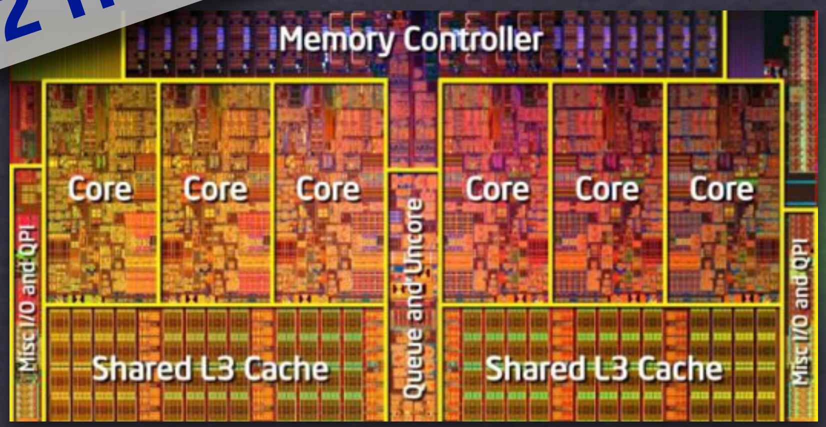
NVIDIA GF100
GPU

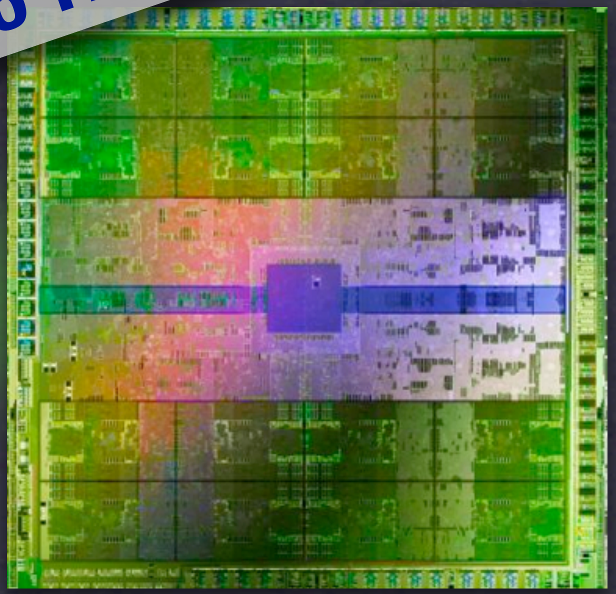
Coarse-grained Versus Fine-grained Parallelism

GPUs require careful program tuning

12 THREADS

24,576 THREADS





Core i7 970
CPU

NVIDIA GF100
GPU

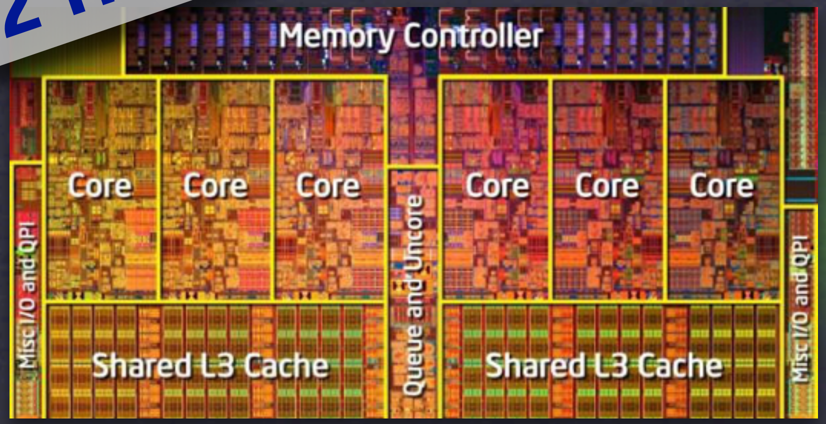
- **SIMD:** groups of threads executing in lock step (warps)
- Need to be careful about control flow

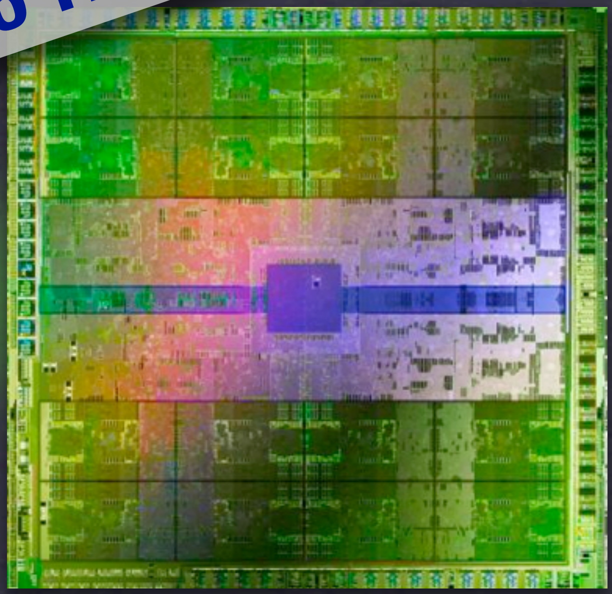
Coarse-grained Versus Fine-grained Parallelism

GPUs require careful program tuning

12 THREADS

24,576 THREADS





Core i7 970
CPU

NVIDIA GF100
GPU

- **SIMD:** groups of threads executing in lock step (warps)
- Need to be careful about control flow

- **Latency hiding:** optimised for regular memory access patterns
- Optimise memory access

Coarse-grained Versus Fine-grained Parallelism

GPUs require careful program tuning

Code generation for embedded code

- Embedded DSL
  - Restricted control flow
  - First-order GPU code
- Generative approach based on combinator templates

Code generation for embedded code

- Embedded DSL
  - Restricted control flow
  - First-order GPU code
- Generative approach based on combinator templates

✓ limited control structures

PLS

Code generation for embedded code

- Embedded DSL
  - Restricted control flow
  - First-order GPU code
- Generative approach based on combinator templates

✓ limited control structures

✓ hand-tuned access patterns

[DAMP 2011]

PLS

Embedded GPU computations

Dot product

```
dotp :: Vector Float -> Vector Float
     -> Acc (Scalar Float)
dotp xs ys
  = let
      xs' = use xs
      ys' = use ys
    in
    fold (+) 0 (zipWith (*) xs' ys')
```

Embedded GPU computations

Dot product

Haskell array

```
dotp :: Vector Float -> Vector Float
     -> Acc (Scalar Float)
dotp xs ys
  = let
      xs' = use xs
      ys' = use ys
    in
    fold (+) 0 (zipWith (*) xs' ys')
```

PLS

Embedded GPU computations

Haskell array

Dot product

```
dotp :: Vector Float -> Vector Float
     -> Acc (Scalar Float)
dotp xs ys
  = let
      xs' = use xs
      ys' = use ys
    in
    fold (+) 0 (zipWith (*) xs' ys')
```

Embedded array = desc. of array comps

PLS

Embedded GPU computations

Haskell array

Dot product

```
dotp :: Vector Float -> Vector Float
     -> Acc (Scalar Float)
dotp xs ys
  = let
      xs' = use xs
      ys' = use ys
    in
    fold (+) 0 (zipWith (*) xs' ys')
```

Embedded array = desc. of array comps

Lift Haskell arrays into EDSL — may trigger host➙device transfer

Embedded GPU computations

Dot product

```
dotp :: Vector Float -> Vector Float
     -> Acc (Scalar Float)
dotp xs ys
  = let
      xs' = use xs
      ys' = use ys
    in
    fold (+) 0 (zipWith (*) xs' ys')
```

Haskell array

Embedded array = desc. of array comps

Lift Haskell arrays into EDSL — may trigger host➙device transfer

Embedded array computations

PLS

Nested Data Parallelism

Modularity

- Standard (Fortran, CUDA, etc.) is flat, regular parallelism
- Same for our libraries for functional data parallelism for multicore CPUs (Repa) and GPUs (Accelerate)
- But we want more…

```
smvm :: SparseMatrix -> Vector -> Vector
smvm sm v = [: sumP (dotp sv v) | sv <- sm :]
```

Parallel array comprehension

```
smvm :: SparseMatrix -> Vector -> Vector
smvm sm v = [: sumP (dotp sv v) | sv <- sm :]
```

```
smvm :: SparseMatrix -> Vector -> Vector
smvm sm v = [: sumP (dotp sv v) | sv <- sm :]
```

Parallel array comprehension

Parallel reduction/fold

Parallel array comprehension

```
smvm :: SparseMatrix -> Vector -> Vector
smvm sm v = [: sumP (dotp sv v) | sv <- sm :]
```

Parallel reduction/fold

Nested parallelism!

```
smvm :: SparseMatrix -> Vector -> Vector
smvm sm v = [: sumP (dotp sv v) | sv <- sm :]
```

Parallel array comprehension

Parallel reduction/fold

Defined in a library: Internally parallel?

Nested parallelism!

```
smvm :: SparseMatrix -> Vector -> Vector
smvm sm v = [: sumP (dotp sv v) | sv <- sm :]
```

Parallel array comprehension

Parallel reduction/fold

Defined in a **library**: Internally parallel?

Nested parallelism!

Nested parallelism?

```
smvm :: SparseMatrix -> Vector -> Vector
smvm sm v = [: sumP (dotp sv v) | sv <- sm :]
```

Parallel array comprehension

Parallel reduction/fold

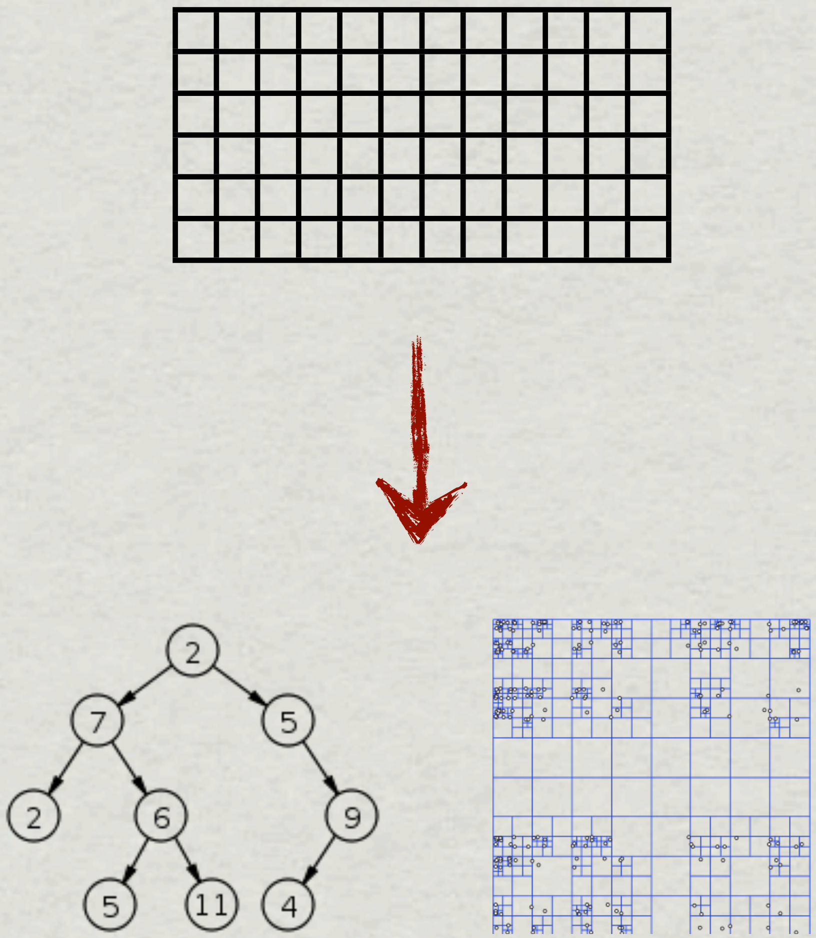
Defined in a **library**: Internally parallel?

Nested parallelism!

Nested parallelism?

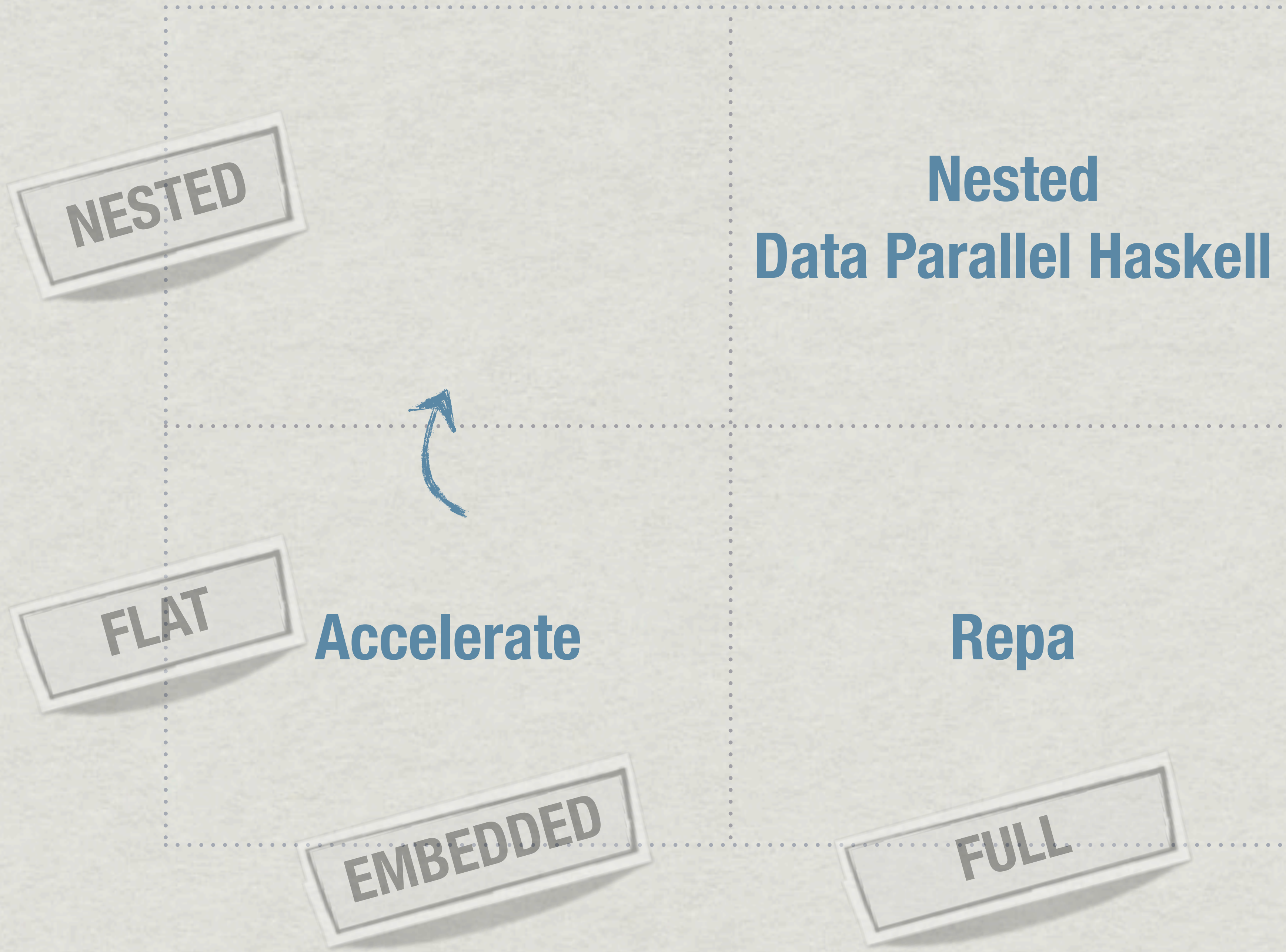
Regular, flat data parallelism is not sufficient!

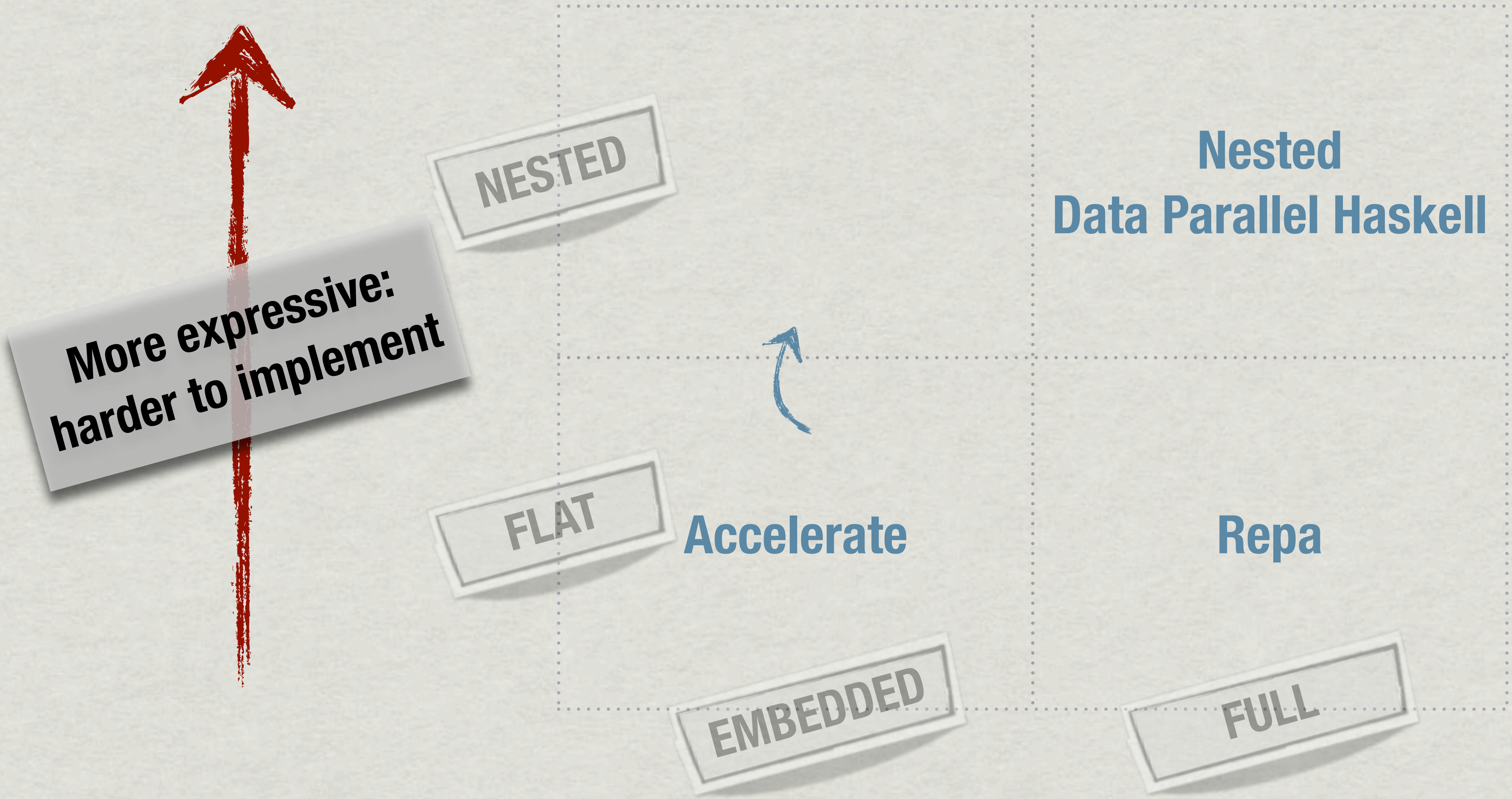
Nested parallelism



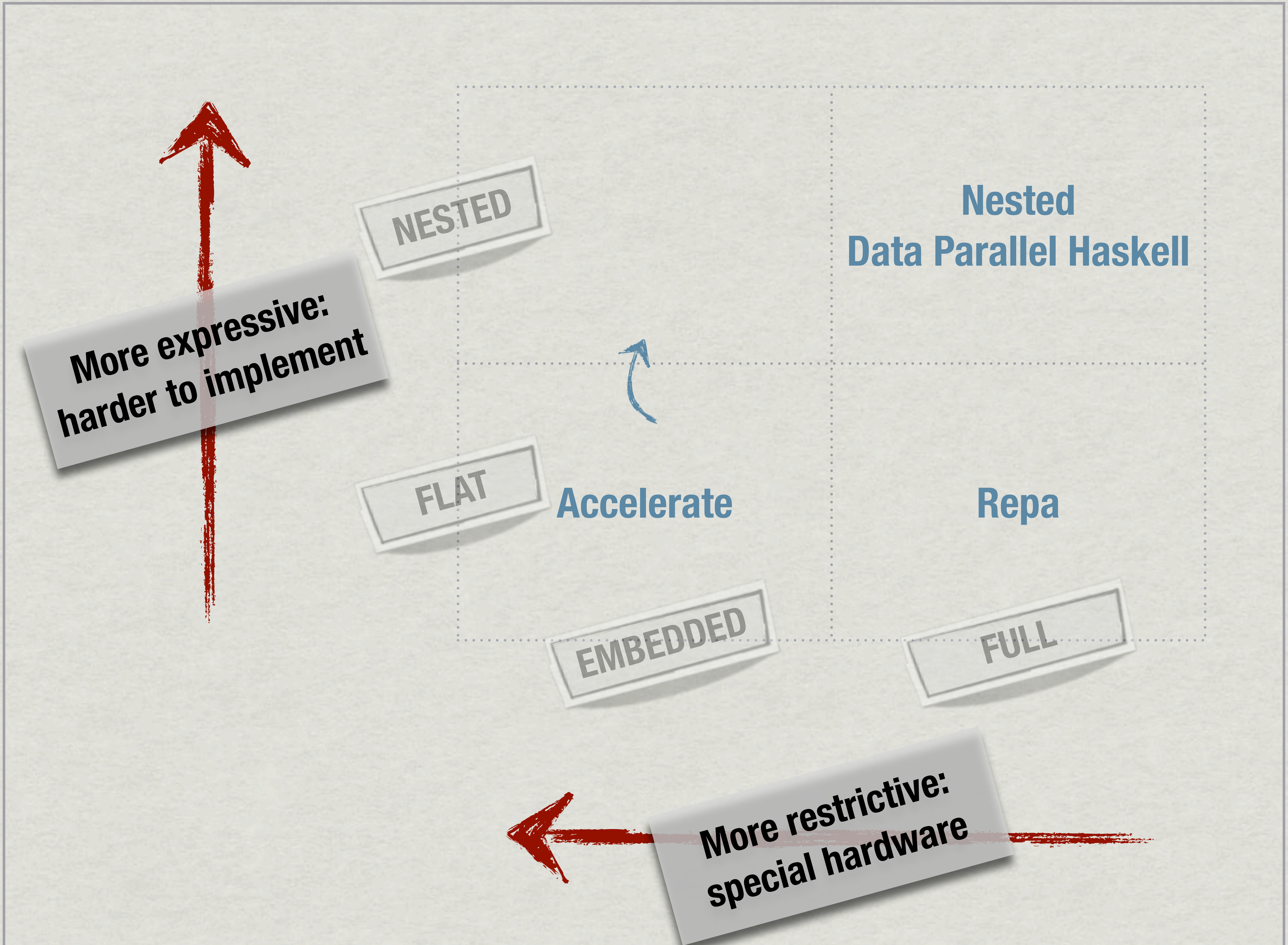
- Modular
- Irregular, nested data structures
  - Sparse structures, tree structures
  - Hierachrchical decomposition
- Nesting to arbitrary, dynamic depth: divide & conquer
- Lots of compiler work: still very experimental! [FSTTCS 2008, ICFP 2012, Haskell 2012b]

PLS


NESTED
Nested
Data Parallel Haskell
FLAT
Accelerate
Repa
EMBEDDED
FULL



More expressive:
harder to implement
NESTED
FLAT
EMBEDDED
FULL
Nested
Data Parallel Haskell
Accelerate
Repa


NESTED
Nested
Data Parallel Haskell
More expressive:
harder to implement
FLAT
Accelerate
Repa
EMBEDDED
FULL
More restrictive:
special hardware

Types are at the centre of everything we are doing

Types are at the centre of everything we are doing

- Types separate pure from effectful code

Types are at the centre of everything we are doing

- Types separate pure from effectful code
- Types guide operational behaviour (data representation, use of parallelism, etc.)

Types are at the centre of everything we are doing

- Types separate pure from effectful code
- Types guide operational behaviour (data representation, use of parallelism, etc.)
- Types identify restricted code for specialised hardware, such as GPUs

PLS

Types are at the centre of everything we are doing

- Types separate pure from effectful code
- Types guide operational behaviour (data representation, use of parallelism, etc.)
- Types identify restricted code for specialised hardware, such as GPUs
- Types guide parallelising program transformations

PLS

Summary

Blog: http://justtesting.org/
Twitter: @TacticalGrace

- Core ingredients
  - Control purity, not concurrency
  - Types guide representations and behaviours
  - Bulk-parallel operations
- Get it
  - Latest Glasgow Haskell Compiler (GHC)
  - Repa, Accelerate & DPH packages from Hackage (Haskell library repository)

PLS

Thank you!

This research has in part been funded by the Australian Research Council and by Microsoft Corporation.

[EuroPar 2001] *Nepal -- Nested Data-Parallelism in Haskell*. Chakravarty, Keller, Lechtchinsky & Pfannenstiel. In "Euro-Par 2001: Parallel Processing, 7th Intl. Euro-Par Conference", 2001.

[FSTTCS 2008] *Harnessing the Multicores: Nested Data Parallelism in Haskell*. Peyton Jones, Leshchinskiy, Keller & Chakravarty. In "IARCS Annual Conf. on Foundations of Software Technology & Theoretical Computer Science", 2008.

[ICFP 2010] *Regular, shape-polymorphic, parallel arrays in Haskell*. Keller, Chakravarty, Leshchinskiy, Peyton Jones & Lippmeier. In Proceedings of "ICFP 2010 : The 15th ACM SIGPLAN Intl. Conf. on Functional Programming", 2010.

[DAMP 2011] *Accelerating Haskell Array Codes with Multicore GPUs*. Chakravarty, Keller, Lee, McDonell & Grover. In "Declarative Aspects of Multicore Programming", 2011.

[Haskell 2011] *Efficient Parallel Stencil Convolution in Haskell*. Lippmeier & Keller. In Proceedings of "ACM SIGPLAN Haskell Symposium 2011", ACM Press, 2011.

[ICFP 2012] *Work Efficient Higher-Order Vectorisation*. Lippmeier, Chakravarty, Keller, Leshchinskiy & Peyton Jones. In Proceedings of "ICFP 2012 : The 17th ACM SIGPLAN Intl. Conf. on Functional Programming", 2012. Forthcoming.

[Haskell 2012a] *Guiding Parallel Array Fusion with Indexed Types*. Lippmeier, Chakravarty, Keller & Peyton Jones. In Proceedings of "ACM SIGPLAN Haskell Symposium 2012", ACM Press, 2012. Forthcoming.

[Haskell 2012b] *Vectorisation Avoidance*. Keller, Chakravarty, Lippmeier, Leshchinskiy & Peyton Jones. In Proceedings of "ACM SIGPLAN Haskell Symposium 2012", ACM Press, 2012. Forthcoming.