Utrecht University

B3CC: Concurrency

11: Accelerate

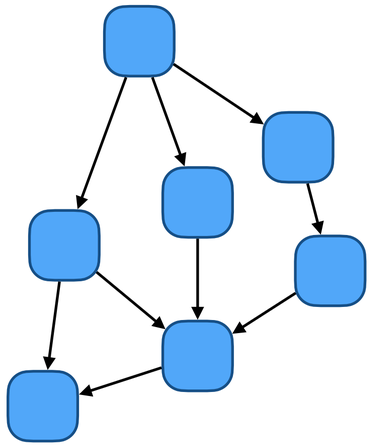
Tom Smeding

Announcement

- The third practical is now available
  - Due Friday 23 January 2026 @ 23:59
  - You may work in pairs
  - Double-check the installation instructions

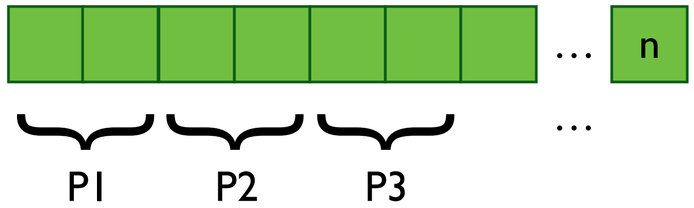
Data parallelism, GPU programming

Recap



Task parallelism

- Explicit threads
- Synchronise via locks, messages, or STM
- Modest parallelism
- Easy to program?



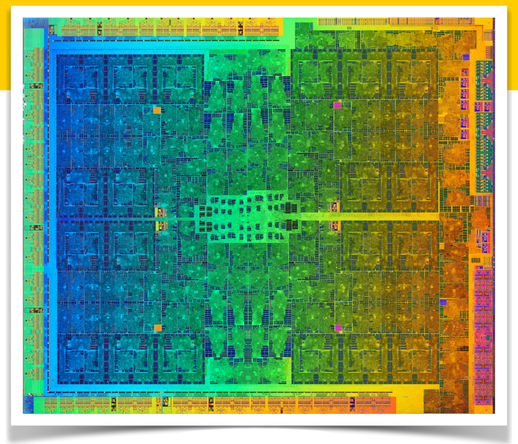
Data parallelism

- Operate simultaneously on bulk data
- Implicit synchronisation
- Massive parallelism
- Easy to program?

Recap

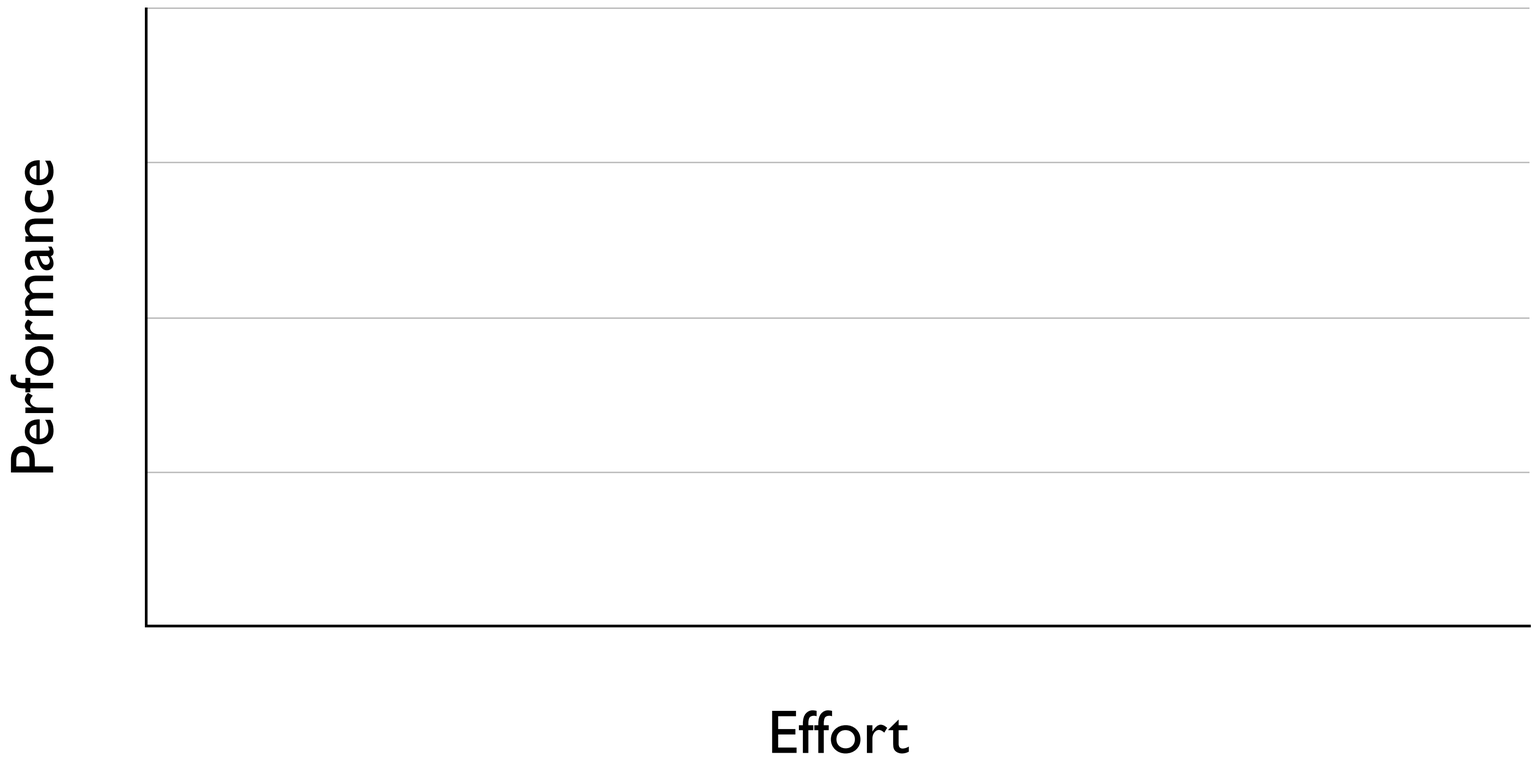
- Despite the name, data parallelism is only a programming model
  - The key is a *single logical thread of control*
  - It does not actually require the operations to be executed in parallel!
  - Today we'll look at a language for data-parallel programming on the GPU

GPU (graphics processing unit)

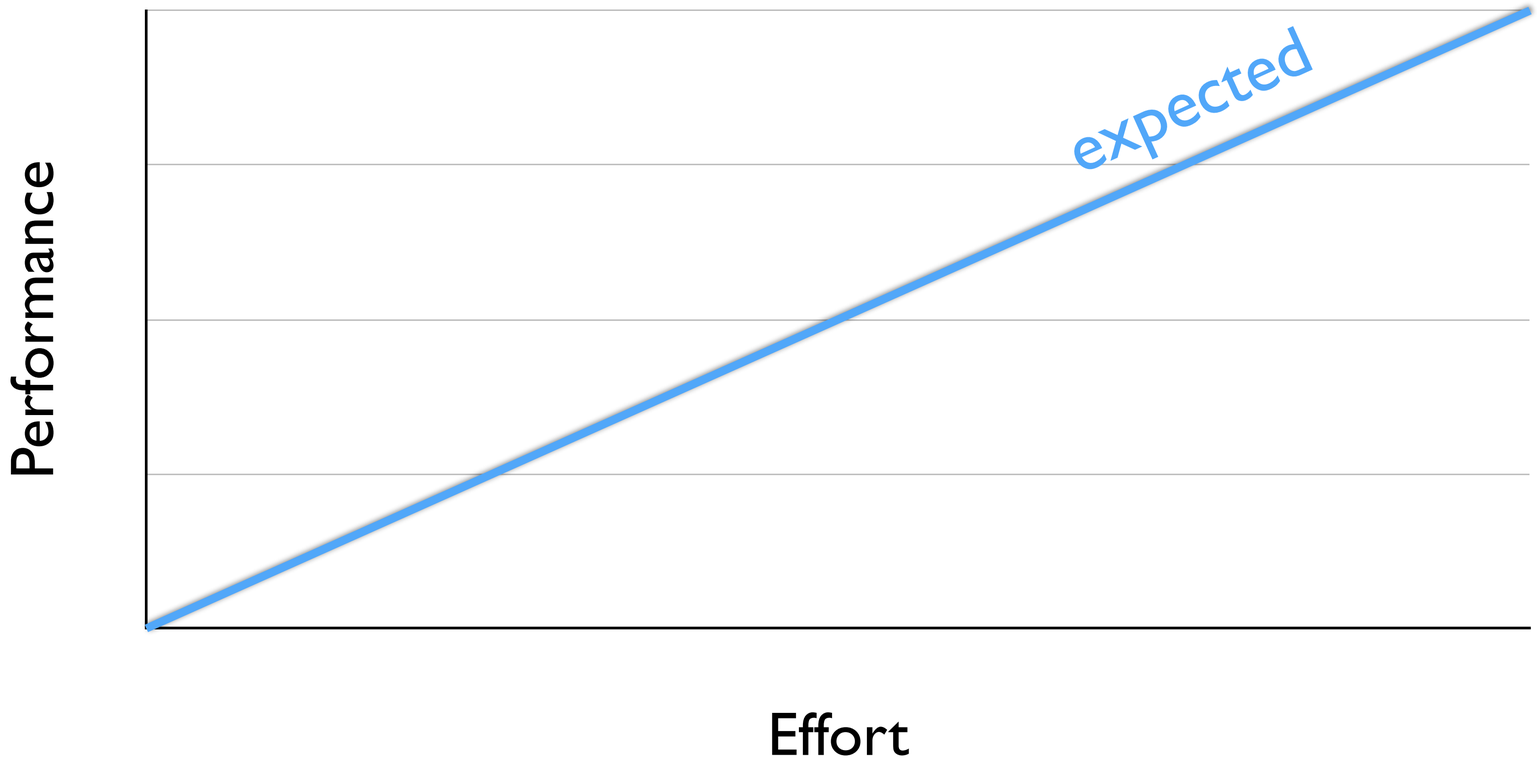


- Lots of interest to use them for non-graphics tasks
  - Machine learning, bioinformatics, data science, weather & climate, medical imaging, computational chemistry, …
  - Can have much higher performance than a traditional CPU
- Specialised hardware with a specialised programming model
  - Memory management is explicit, with distinct memory spaces
  - Thousands of threads running simultaneously, each of which can modify any piece of memory at any time

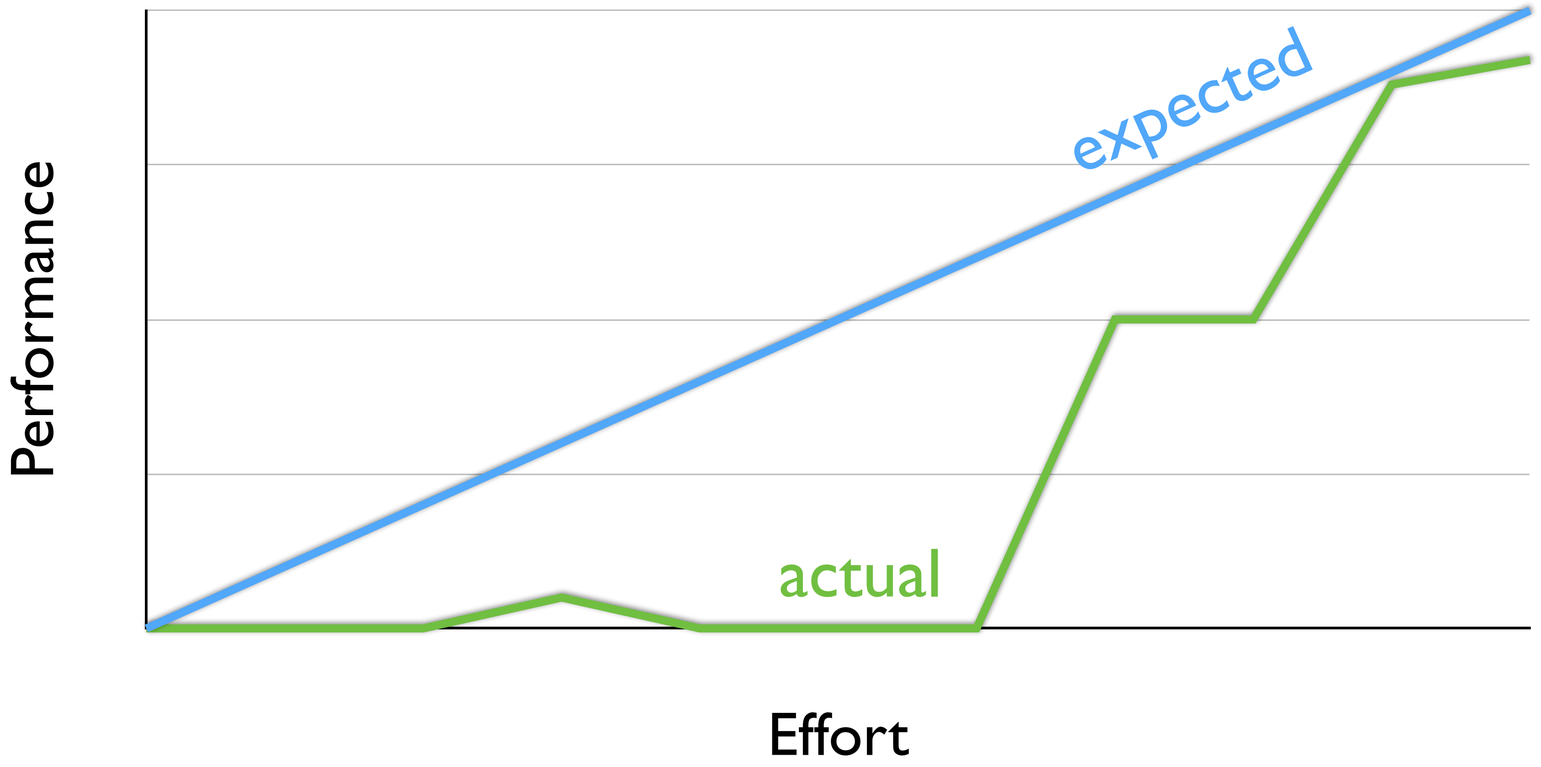
GPU programming



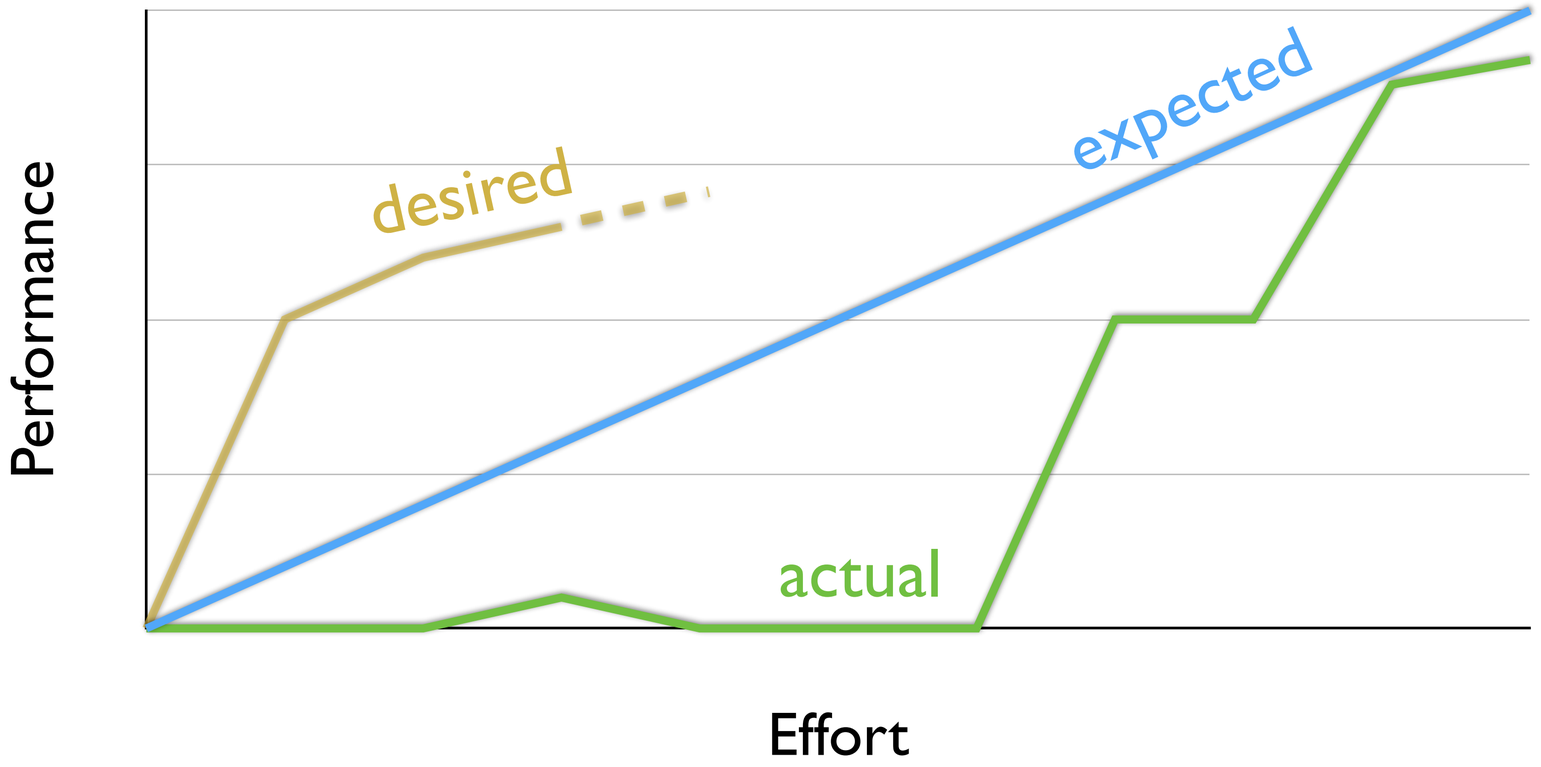
GPU programming



GPU programming

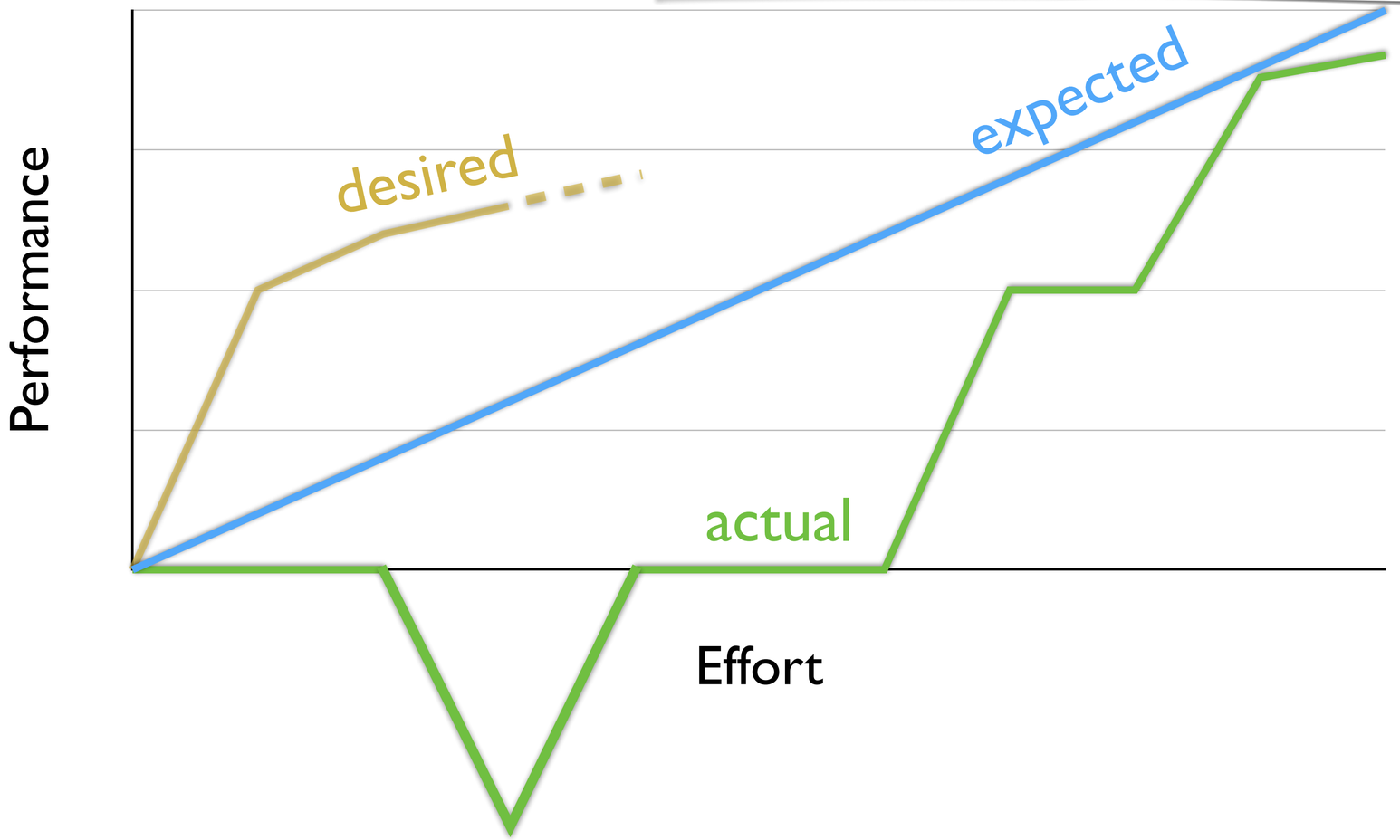


GPU programming



GPU programming

> After expressing available parallelism, I often find that the code has slowed down.
> — Jeff Larkin, NVIDIA Developer Technology



https://devblogs.nvidia.com/getting-started-openacc/

GPU programming

- Two main difficulties:
  1. Structuring the program in a way suitable for GPU parallelisation ←
  2. Writing (performant) GPU code

Accelerate

Accelerate

- An embedded language for *data-parallel arrays* in Haskell
  - Takes care of generating the high-performance CPU/GPU code for us
  - Computations take place on dense multi-dimensional arrays
  - Parallelism is introduced in the form of collective operations on arrays
    - map, zipWith, fold, scan (various kinds); permutations (data movement); etc.
    - It is a *restricted* language: consists only of operations which can be executed efficiently in parallel

Example: dot product

- In Haskell (lists):

```
import Prelude

dotp :: Num a
     => [a]
     -> [a]
     -> a
dotp xs ys = foldl' (+) 0 (zipWith (*) xs ys)
```

Example: dot product

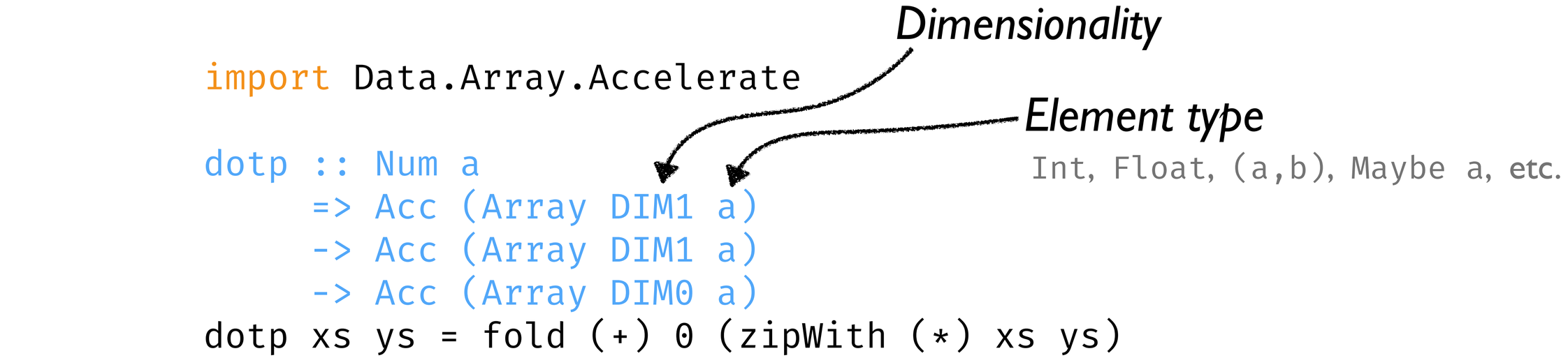
- In Accelerate:

```
import Data.Array.Accelerate

dotp :: Num a
     => Acc (Vector a)
     -> Acc (Vector a)
     -> Acc (Scalar a)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

Example: dot product

- In Accelerate:




```
import Data.Array.Accelerate

dotp :: Num a
     => Acc (Array DIM1 a)
     -> Acc (Array DIM1 a)
     -> Acc (Array DIM0 a)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

```
Scalar a = Array DIM0 a = Array Z a
Vector a = Array DIM1 a = Array (Z :. Int) a
Matrix a = Array DIM2 a = Array (Z :. Int :. Int) a
           Array DIM3 a = Array (Z :. Int :. Int :. Int) a
              ...
```

Accelerate

- Compile and execute an Accelerate program
  - The same program can be run on different targets

```
import Data.Array.Accelerate.Interpreter
-- import Data.Array.Accelerate.LLVM.Native
-- import Data.Array.Accelerate.LLVM.PTX

run  :: (…) => Acc a -> a
run1 :: (…) => (Acc a -> Acc b) -> a -> b

runN :: Afunction f => f -> AfunctionR f
runN :: (…) => Acc a -> a
runN :: (…) => (Acc a -> Acc b) -> a -> b
runN :: (…) => (Acc a -> Acc b -> Acc c) -> a -> b -> c
-- ...
```

There's also `runQ`, but don't worry about that

Accelerate

- Parallel computations take place on arrays
  - A *stratified* language of parallel (`Acc`) and scalar (`Exp`) computations
  - Parallel operations consist of many scalar expressions executed in parallel

Accelerate

- The `map` operation:
  - A collective operation (`Acc`) which applies the given scalar function (`Exp`) to each element of the array *in parallel*
  - `map (\x -> x+1) xs` on a one-dimensional array of floats:

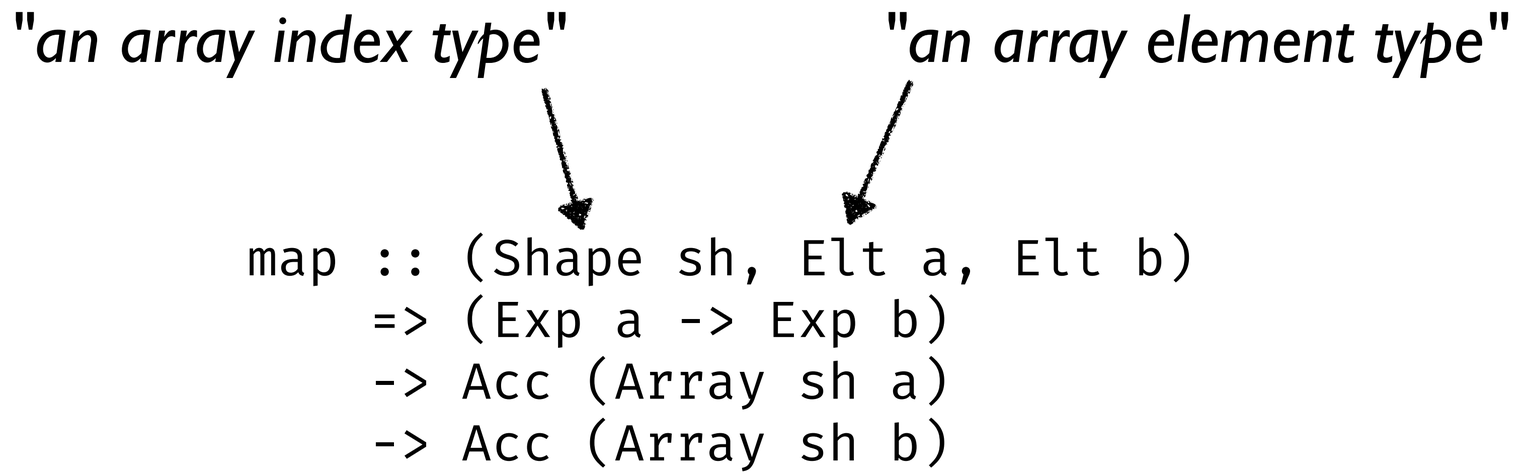
Acc

```
__global__ void map(const float* d_xs, float* d_ys, int len)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < len) {
        float x = d_xs[i];
        d_ys[i] = x + 1;
    }
}
```

Exp

Accelerate

- The `map` operation:
  - A collective operation (`Acc`) which applies the given scalar function (`Exp`) to each element of the array *in parallel*
  - `map (\x -> x+1) xs` on a one-dimensional array of floats:



Oddities

- Accelerate is a language *embedded* in Haskell
  - We reuse much of the syntax, but the semantics are different
    - Strict evaluation, unboxed data, no general recursion…
  - Actually, `Acc` and `Exp` are just data structures!
    - Have a `Show` instance
    - The Haskell program *generates* the Accelerate program
    - The `run` operation performs runtime (cross) compilation
  - But the integration has some oddities as well…

Lifting & Unlifting

- Consider the following two types:

```
x :: (Exp Int, Exp Int)
y :: Exp (Int, Int)
```

  - The first is a Haskell pair of embedded expressions on `Int`
  - The second is an embedded expression returning a pair of `Int`s
- How to convert between the two?
  - The pattern synonym `T2`
  - (legacy: the functions `lift` and `unlift` (not recommended))

Pattern synonyms

- We use pattern synonyms for constructing & destructing embedded tuples
  - Can't overload built-in syntax `( , )`, `( , , )`, etc.
  - Instead we use `T2`, `T3`, etc. at both the `Acc` and `Exp` level

```
result :: Acc (Vector Int, Scalar Int)
result = …

T2 idx tot = result
  -- idx :: Acc (Vector Int)
  -- tot :: Acc (Scalar Int)

res = T2 tot idx
  -- res :: Acc (Scalar Int, Vector Int)
```

Shapes

- Array shapes (& indices) are snoc-lists formed from `Z` and `(:.)`
  - `Z` is a zero-dimensional (scalar)
  - `(:.)` adds one inner-most dimension on the right

```
type DIM1     = Z :. Int
type Vector a = Array DIM1 a
```

- More pattern synonyms for constructing & destructing indices

```
x       :: Exp Int
I1 x    :: Exp DIM1    -- you'll need this one
I2 x y  :: Exp DIM2
```

Pattern matching

- Use the `match` operator to perform pattern matching in embedded code
  - Also note the pattern synonyms for constructing/deconstructing cases

```
foo :: Exp (Maybe Int) -> Exp Int
foo x = x & match \case
  Nothing_ -> 0
  Just_ y  -> y + 1
```

Guards

- Unfortunately guard syntax doesn't work
  - Use a regular if-then-else (chain) instead

```
nope :: Exp Int -> Exp Int
nope x
  | x < 0      = ...
  | otherwise = ...
```

```
okay1 :: Exp Int -> Exp Int
okay1 x = cond (x < 0)
               (...)
               (...)
```

```
okay2 :: Exp Int -> Exp Int
okay2 x = if x < 0
            then ...
            else ...
```

Looping

- Can’t write recursive embedded functions directly
  - Need to use an explicit looping combinator instead
  - Continue applying the body function (second argument) as long as the predicate function (first argument) returns true

```
awhile
  :: Arrays a
  => (Acc a -> Acc (Scalar Bool))
  -> (Acc a -> Acc a)
  -> Acc a
  -> Acc a
```

Debugging

- Some trace functions for printf-style debugging
  - Output a trace message as well as some arrays to the console before proceeding with the computation
  - Useful for inspecting intermediate values

```
atraceArray
  :: (Arrays a, Arrays b)
  => Text
  -> Acc a
  -> Acc b
  -> Acc b
```

use `"quotes"`

```
import Data.Array.Accelerate.Debug.Trace
```

Documentation

- More information in the documentation
  - https://ics-websites.science.uu.nl/docs/vakken/b3cc/resources/acc-head-docs/ (latest version used in the Quickhull template)

Accelerate

- Implementing a data-parallel program consists of two parts:
  - What are the collective (parallel) operations that need to be done?
  - What does each individual (sequential) thread need to do?

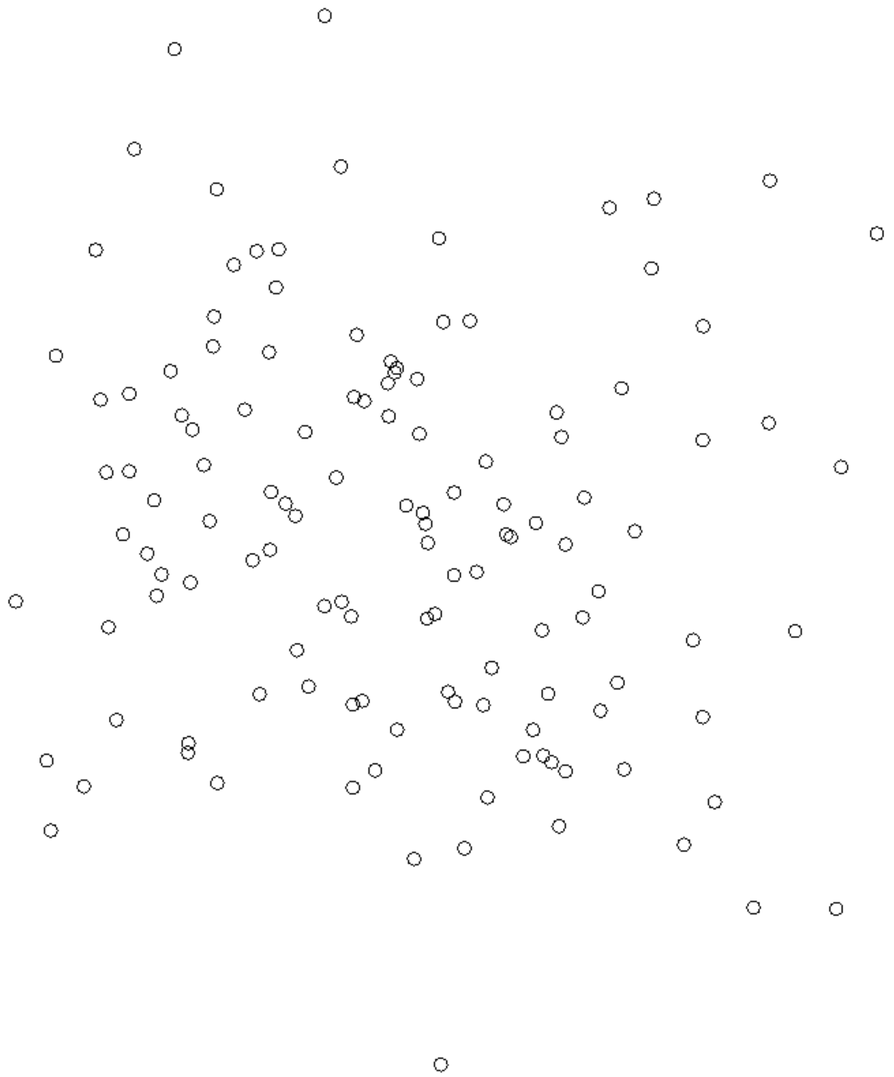
Quickhull

Quickhull

- An algorithm to determine the smallest polygon that contains a set of points
  - You will implement a data-parallel version of the algorithm in Accelerate
  - See the specification for details

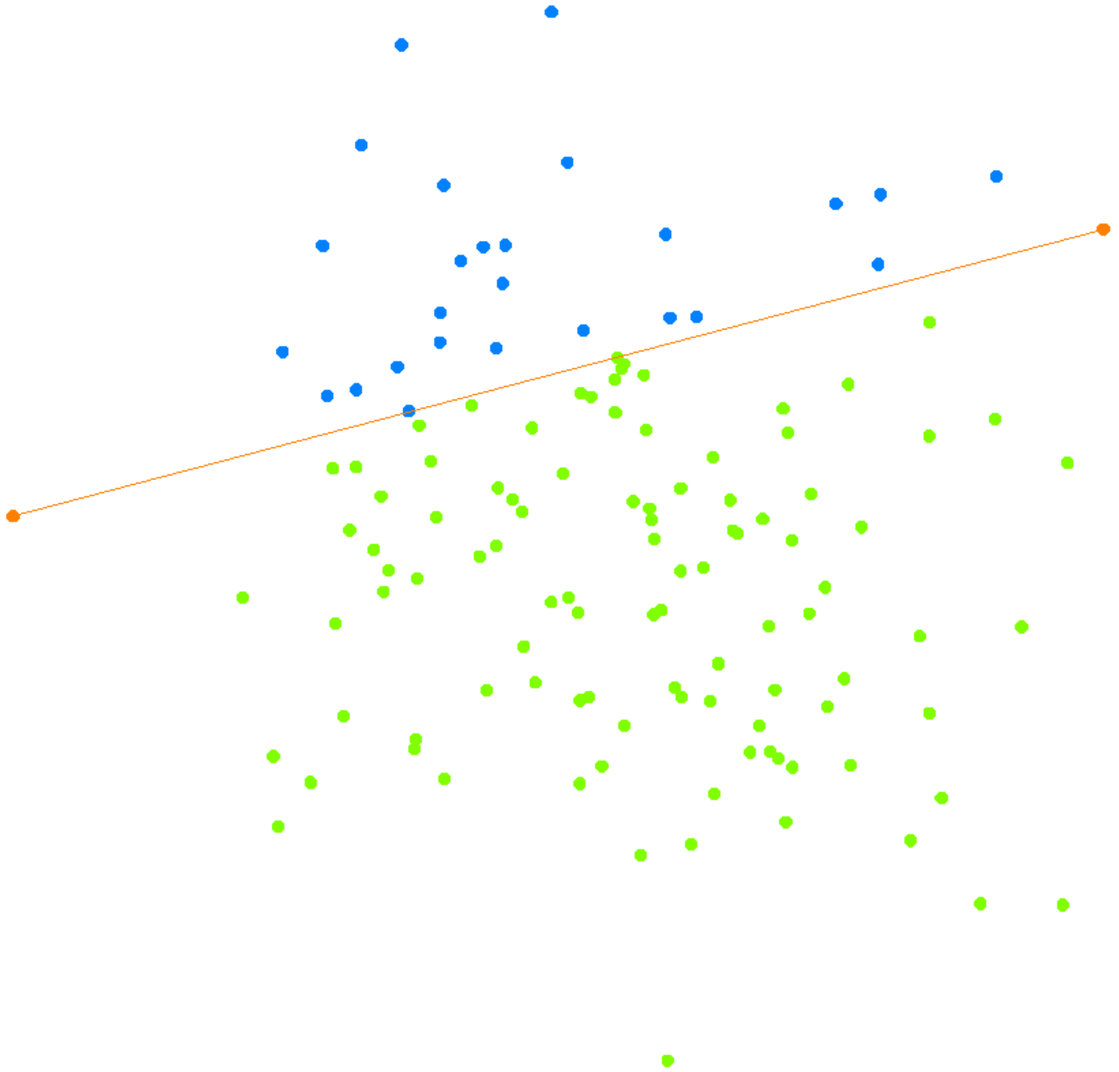
Example

- Initial points
  - The goal is to find the smallest polygon containing all these points
  - This is known as the *convex hull*



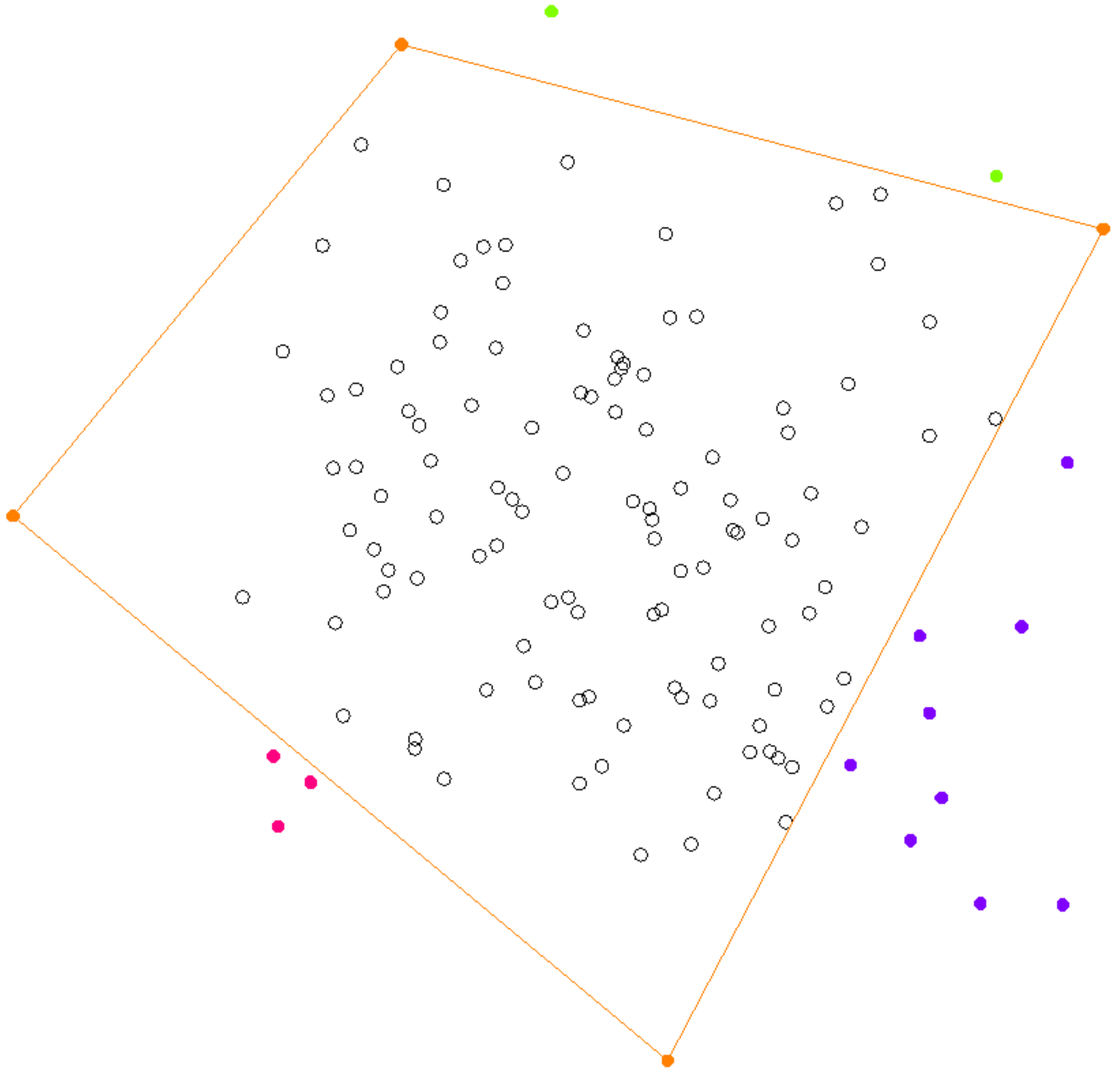
Example

- Create initial partition
  - Choose two points that are definitely on the convex hull
  - Partition others to either side of that line (above/left and below/right)
  - Points of the same colour are in the same segment



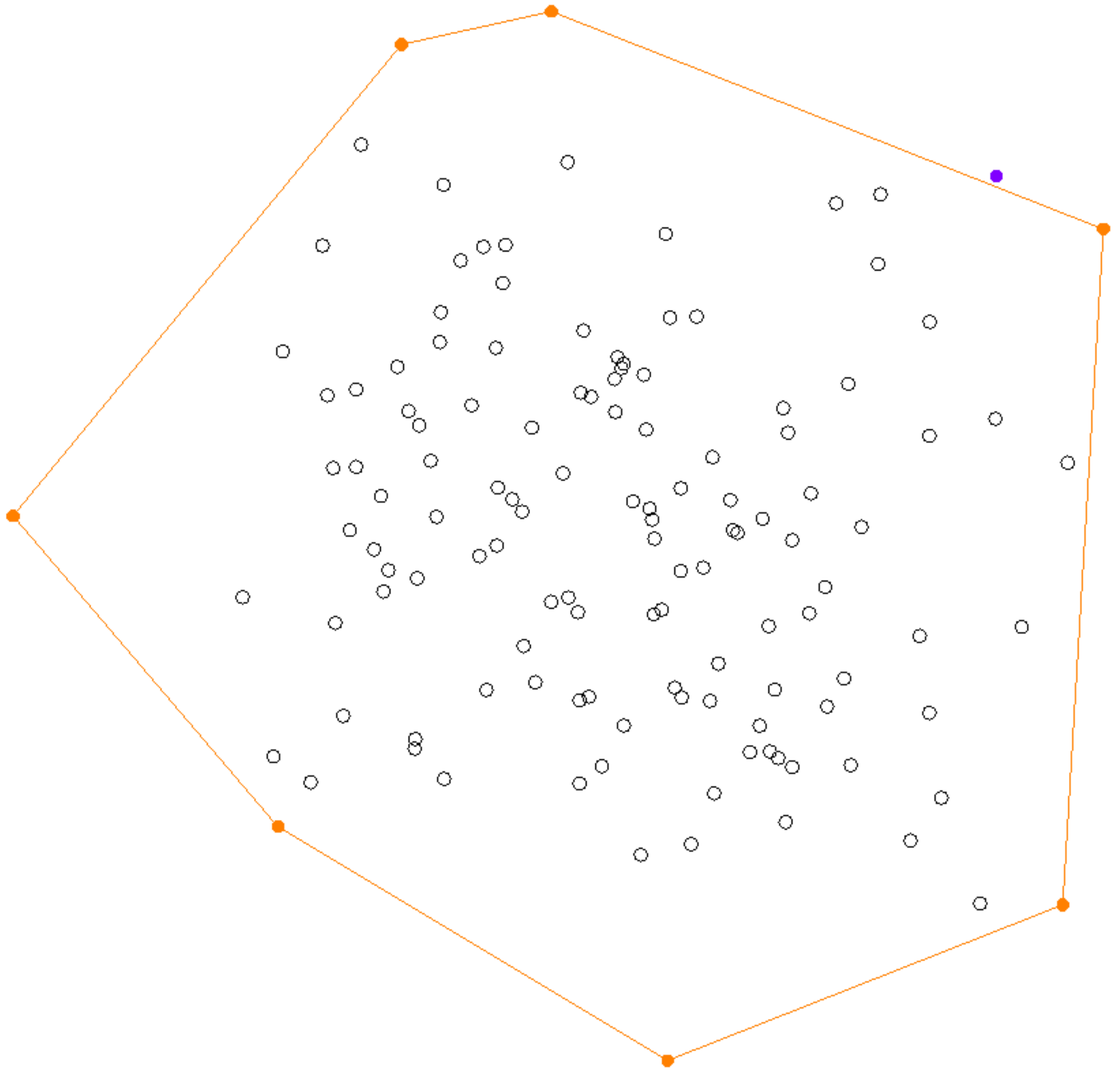
Example

- Recursively partition each segment
  - This is done for all points at once, in data-parallel
  - The hollow circles are points no longer under consideration
  - Orange circles are on the convex hull
  - Other colours are still undecided.
  - Same colours are in the same partition



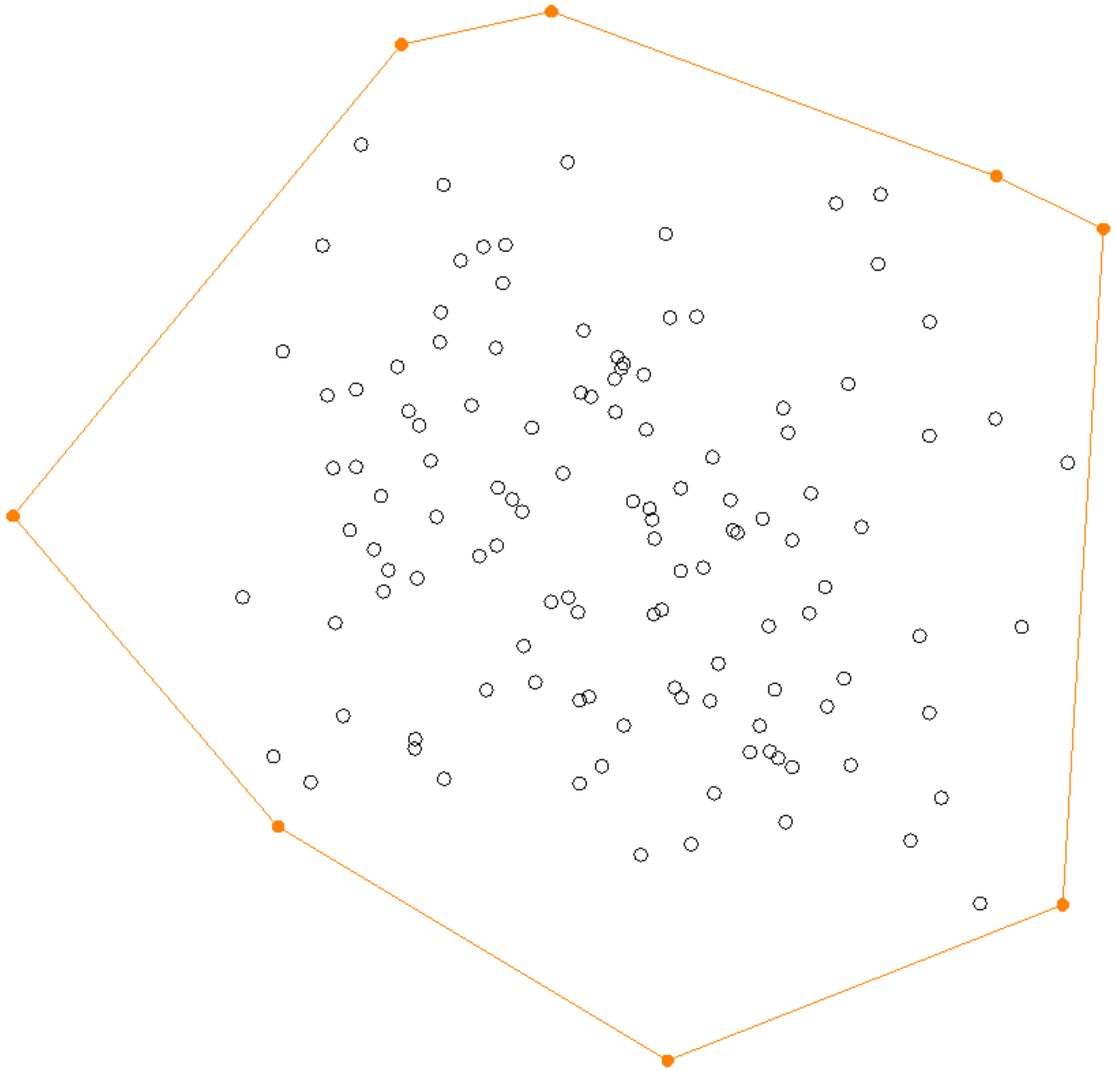
Example

- Continue partitioning each segment…



Example

- … until no undecided points remain



Extra slides

Traditional compiler construction

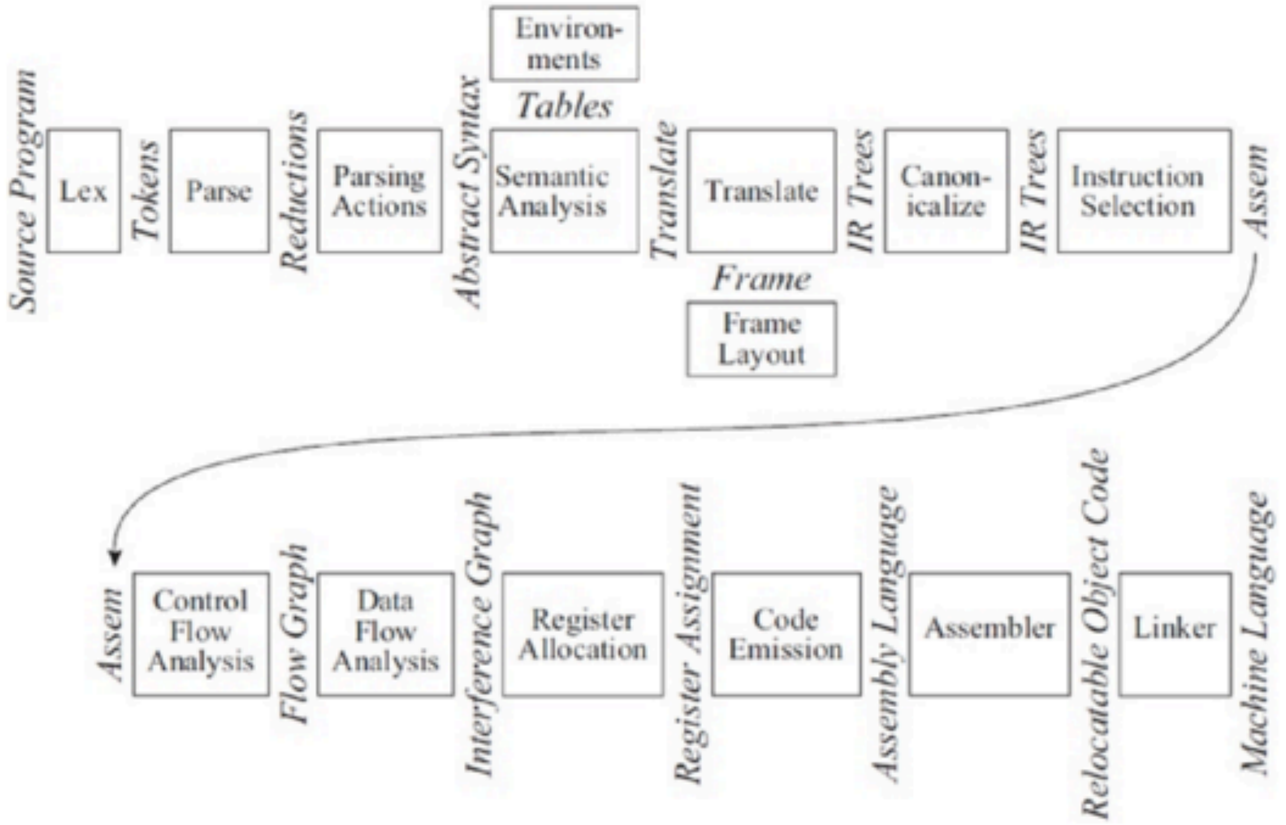


FIGURE 1.1. Phases of a compiler, and interfaces between them

Modern compiler construction

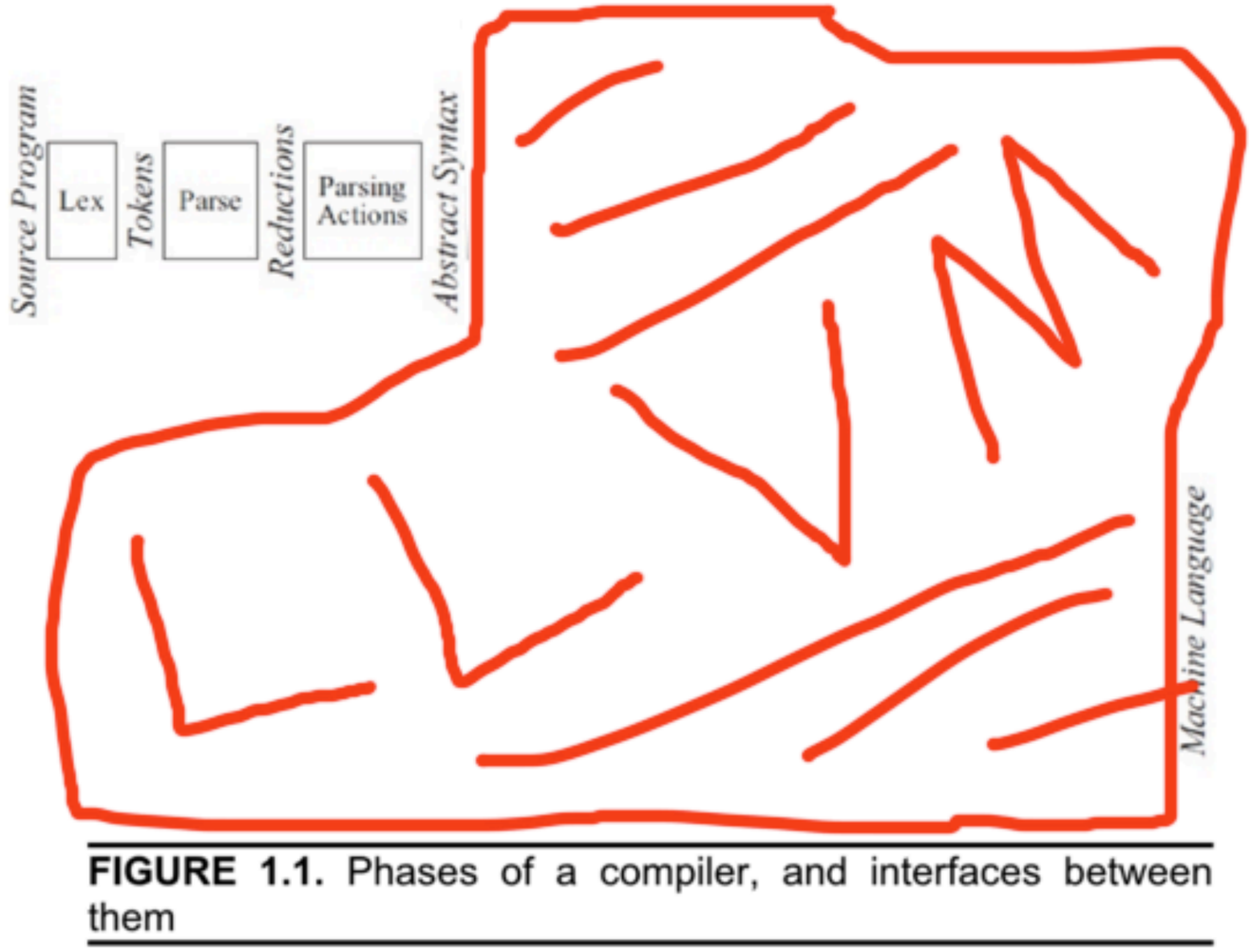


FIGURE 1.1. Phases of a compiler, and interfaces between them

https://msm.runhello.com/p/1003