An introduction to deterministic parallel programming and data parallelism with NumPy and Futhark

Michael Kirkedal Thomsen

Based on slides by Troels Henriksen (athas@sigkill.dk)

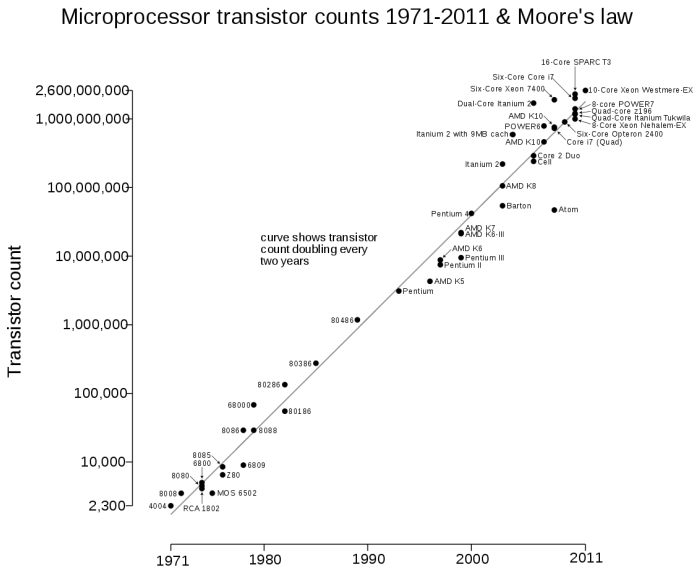
April 28, 2022

Content and motivation

Array programming

Higher-order array programming

The situation so far



Moore's Law

> *The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. –Gordon Moore, 1965*

Note: not actually a law, and frequently misinterpreted.

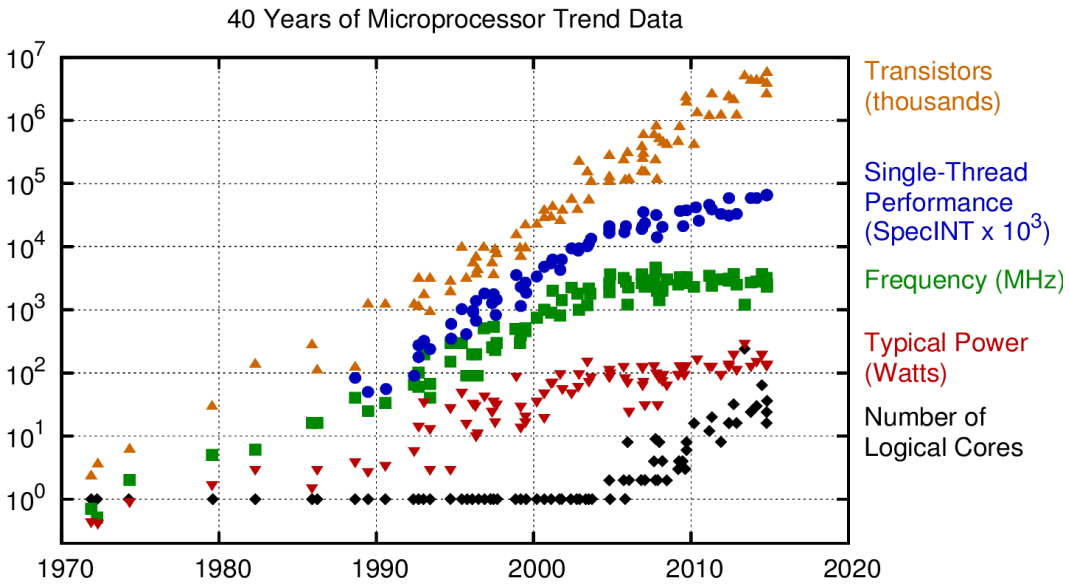
Moore's Law

> *The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. –Gordon Moore, 1965*

Note: not actually a law, and frequently misinterpreted.

- Improvements in transistor fabrication technology resulted in smaller transistors.
  - More transistors for the same chip area.
- This was also translated into higher clock frequencies.

CPU trends over 40 years



CPU trends over 40 years

- Physical limitations stopped most increases in CPU clock frequency around 2005. Instead, the still-continuing increases in transistor counts was used to make multicore CPUs.
- We had to program with a little more parallelism, but within each thread, we still had a standard CPU model.
- Now even Moore's Law is beginning to end, and we must make better use of the transitors we have—we won't get that many more.
- This has made exotic non-CPU architectures more attractive.

Two kinds of parallelism

Task parallelism

Simultaneously performing *different* operations on potentially different pieces of data. Often nondeterministic.

Two kinds of parallelism

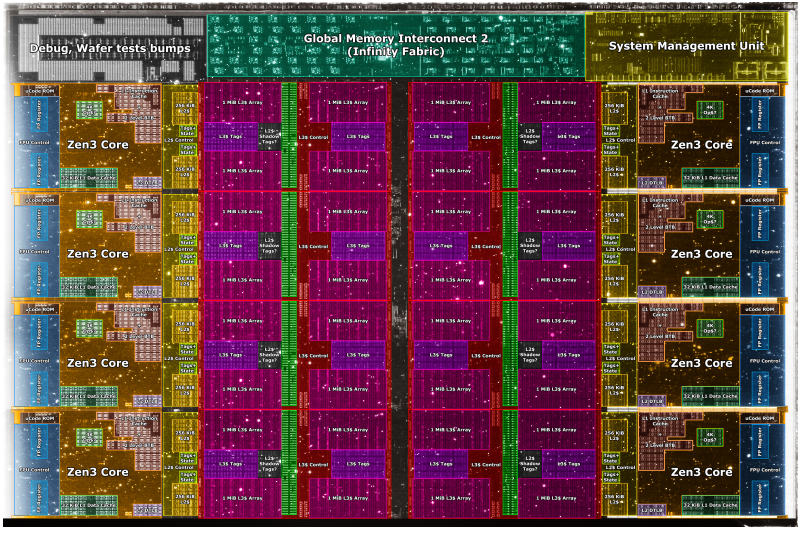
Task parallelism

Simultaneously performing *different* operations on potentially different pieces of data. Often nondeterministic.

Data parallelism

Simultaneously performing *the same* operation on different pieces of the same data. Almost always deterministic.

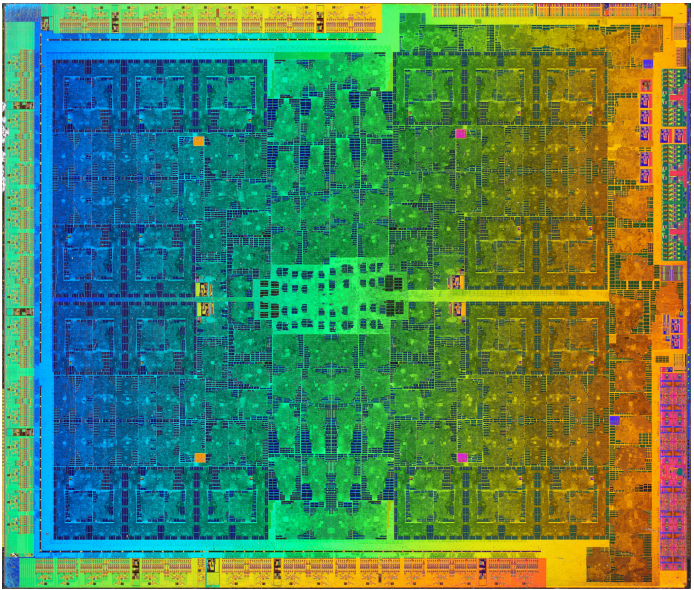
AMD Ryzen 3



https://www.reddit.com/r/Amd/comments/jqjg8e

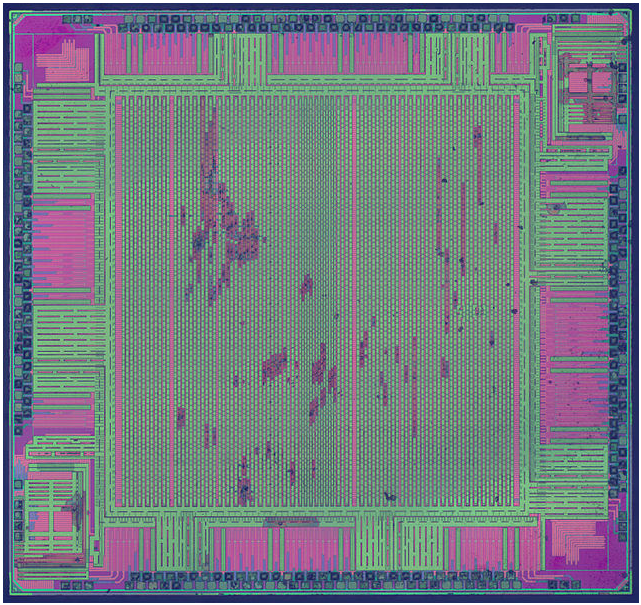
- Irregular layout with many parts.
- Most transitors spent on OOO execution, speculation, caches, branch prediction, etc.
- Few transistors spent on actual computation.
- Task parallel. Fairly easy to program.

NVIDIA Pascal GPU



- Layout consists of replicated components.
- Most transistors spent on computation.
- Data parallel. Quite hard to program.

Altera Cyclone 1 FPGA



- Extremely regular layout.
- Not cleanly task- or data parallel. Extremely hard to program.

The simplified tradeoff

For a fixed transistor budget, a hardware designer can choose to spend transistors on

1. Making the chip more flexible and easier to program.
2. Improving the peak computational capacity.

Modern CPUs are heavily invested in (1).

The simplified tradeoff

For a fixed transistor budget, a hardware designer can choose to spend transistors on

1. Making the chip more flexible and easier to program.
2. Improving the peak computational capacity.

Modern CPUs are heavily invested in (1).

A silver bullet?

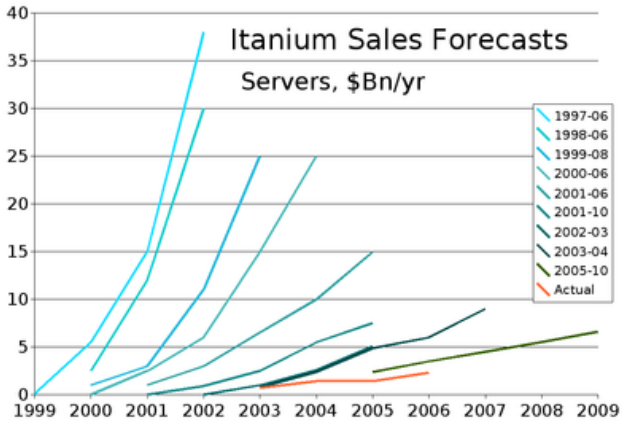
Can we build hardware that is fast and inflexible, and paper over the programming difficulty with a Sufficiently Smart Compiler?

The Itanium

- Intel's *Itanium* was a VLIW 64-bit replacement for x86 designed in the 1990s.
- Assuming sufficient compilers, performed well.

The Itanium

- Intel's *Itanium* was a VLIW 64-bit replacement for x86 designed in the 1990s.
- Assuming sufficient compilers, performed well.



Why didn’t the Itanium work?

The wished-for compilers turned out to be impossible to write.

Why didn’t the Itanium work?

The wished-for compilers turned out to be impossible to write.

1. Bridging the gap from conventional C programs to Itanium’s explicitly parallel EPIC architecture was very hard.
2. Increasing transistor counts meant CPUs could just do their own dynamic superscalar scheduling in hardware.
3. Itanium was outcompeted by ever more complex (soon multicore) x86-64 CPUs that handled legacy code just fine.

Why didn't the Itanium work?

The wished-for compilers turned out to be impossible to write.

1. Bridging the gap from conventional C programs to Itanium's explicitly parallel EPIC architecture was very hard.
2. Increasing transistor counts meant CPUs could just do their own dynamic superscalar scheduling in hardware.
3. Itanium was outcompeted by ever more complex (soon multicore) x86-64 CPUs that handled legacy code just fine.

Important lesson

We cannot expect compilers to map legacy models of programming to new hardware.

The usual compiler disappointments

Probstring's Law

Compiler Advances Double Computing Power Every 18 Years

The usual compiler disappointments

Probstring's Law

Compiler Advances Double Computing Power Every 18 Years

- Empirically true in isolation.
- *Probably not true* in a wider sense, where we use compiler technology to make exotic hardware more accessible.

Content and motivation

Array programming

Higher-order array programming

Data parallelism

Data parallelism

Simultaneously performing *the same* operation on different pieces of the same data.
Almost always deterministic.

Data parallelism

Data parallelism

Simultaneously performing *the same* operation on different pieces of the same data.
Almost always deterministic.

If `x` and `y` are vectors, then `x + y` is a data parallel operation.

Data parallelism

Data parallelism

Simultaneously performing *the same* operation on different pieces of the same data. Almost always deterministic.

If `x` and `y` are vectors, then `x + y` is a data parallel operation.

- **You already know data parallel programming!**
  - NumPy, R, Matlab, Julia, SQL are all data parallel models.
- Completely sequential and deterministic semantics.
- Parallel *execution* straightforward.
- **Distinguish between parallel execution and parallel potential.**
  - Implementations can always be parallelised as long as the code is written in a parallel style.

NumPy

- Python library that provides *n*-dimensional arrays.
- Not usually parallel in practice, but mostly implemented in efficient C and Fortran.
- *Bulk operations* on entire arrays.

```
import numpy as np

def inc_scalar(x):
  for i in range(len(x)):
    x[i] = x[i] + 1

def inc_par(x):
  return x + np.ones(x.shape)
```

- Very often essentially purely functional.
- *First order.* All parallel operations do just one thing. No `map`.

The good news

- NumPy is extremely widely used. This proves empirically that parallel programming does not have to be hard! Let's try some parallel programming.

[1] Troels' informal nomenclature: an operation is "potentially parallel" if it could straightforwardly be executed in parallel, even if a concrete implementation such as Numpy implements it sequentially.

The good news

- NumPy is extremely widely used. This proves empirically that parallel programming does not have to be hard! Let's try some parallel programming.

```
def dotprod_seq(x, y):
    acc = 0
    for i in range(len(x)):
        acc += x[i] * y[i]
    return acc
```

Can we write a *potentially parallel*[1] `dotprod` with NumPy?

[1] Troels' informal nomenclature: an operation is "potentially parallel" if it could straightforwardly be executed in parallel, even if a concrete implementation such as Numpy implements it sequentially.

Just to get it out the way

```
dotprod = np.dot
```

Just to get it out the way

```
dotprod = np.dot
```

Yes, in practice do it this way, but let's see how we could build it from *primitives*.

Parallel dot product

```
def dotprod(x, y):
    products = x * y
    return np.sum(products)
```

```
x                =  [ 1,  2,  -3]
y                =  [ 4,  5,   6]
products         =  [ 4, 10, -18]
np.sum(products) =           -4.0
```

What about that `np.sum`?

Parallel summation

```
def sum_pow2(x):
    while len(x) > 1:
        x_first = x[0:len(x)/2]
        x_last = x[len(x)/2:]
        x = x_first + x_last
    return x[0]
```

Parallel summation

```
def sum_pow2(x):
    while len(x) > 1:
        x_first = x[0:len(x)/2]
        x_last = x[len(x)/2:]
        x = x_first + x_last
    return x[0]
```

```
x       =  [1, 2, 3, 4, 5, 6, 7, 8]
x_first =           [1, 2,  3,  4]
x_last  =           [5, 6,  7,  8]
-------------------------------------
x       =           [6, 8, 10, 12]
x_first =                [ 6,  8]
x_last  =                [10, 12]
-------------------------------------
x       =                [16, 20]
x_first =                    [16]
x_last  =                    [20]
-------------------------------------
x       =                    [36]
```

Wait, is that correct?

In our summation, we changed

$$(((((((x_0 + x_1) + x_2) + x_3) + x_4) + x_5) + x_6) + x_7)$$

to

$$((x_0 + x_4) + (x_2 + x_6)) + ((x_1 + x_5) + (x_3 + x_7))$$

Is that valid?

Wait, is that correct?

In our summation, we changed

$$(((((((x_0 \oplus x_1) \oplus x_2) \oplus x_3) \oplus x_4) \oplus x_5) \oplus x_6) \oplus x_7)$$

to

$$((x_0 \oplus x_4) \oplus (x_2 \oplus x_6)) \oplus ((x_1 \oplus x_5) \oplus (x_3 \oplus x_7))$$

Is that valid?

What about now?

Wait, is that correct?

In our summation, we changed

$$(((((((x_0 \oplus x_1) \oplus x_2) \oplus x_3) \oplus x_4) \oplus x_5) \oplus x_6) \oplus x_7)$$

to

$$((x_0 \oplus x_4) \oplus (x_2 \oplus x_6)) \oplus ((x_1 \oplus x_5) \oplus (x_3 \oplus x_7))$$

Is that valid?

What about now?

In general, what must hold for an operator $\oplus$ for such a rewrite to be valid?

Commutativity and associativity

A binary operator $\oplus : \alpha \to \alpha \to \alpha$ is said to be *commutative* if

$$x \oplus y = y \oplus x$$

Commutativity and associativity

A binary operator $\oplus : \alpha \to \alpha \to \alpha$ is said to be *commutative* if

$$x \oplus y = y \oplus x$$

It is *associative* if

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z$$

If we are a little more clever, we can do a parallel "sum" with any associative operator.

Commutativity and associativity

A binary operator $\oplus : \alpha \to \alpha \to \alpha$ is said to be *commutative* if

$$x \oplus y = y \oplus x$$

It is *associative* if

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z$$

If we are a little more clever, we can do a parallel "sum" with any associative operator.

For convenience, we also tend to require a *neutral element* $0_\oplus$:

$$x \oplus 0_\oplus = 0_\oplus \oplus x = x$$

Monoidal summation

An associative binary operator $\oplus : \alpha \to \alpha \to \alpha$ with a neutral element $0_\oplus$ is called a *monoid* and written $(\oplus, 0_\oplus)$[2]. Examples:

- $(+, 0)$
- $(*, 1)$
- $(++, [])$

[2] A monoid without a neutral element is called a *semigroup*

Monoidal summation

An associative binary operator $\oplus : \alpha \to \alpha \to \alpha$ with a neutral element $0_\oplus$ is called a *monoid* and written $(\oplus, 0_\oplus)$[2]. Examples:

- $(+, 0)$
- $(*, 1)$
- $(++, [])$

For any monoid $(\oplus, 0_\oplus)$ we can construct a function `sum` with *semantics*

$$\texttt{sum}([x_0, \ldots, x_{n-1}]) = x_0 \oplus \ldots \oplus x_{n-1}$$

but which can *operationally* be executed in parallel.

[2] A monoid without a neutral element is called a *semigroup*

Monoidal reduction

We can abstract the notion of summation into a higher-order function typically called `reduce`:

$$\texttt{reduce} : (\alpha \to \alpha \to \alpha) \to \alpha \to []\alpha \to \alpha$$

Then

$$\texttt{sum} = \texttt{reduce}\ (+)\ 0$$

Monoidal reduction

We can abstract the notion of summation into a higher-order function typically called `reduce`:

$$\texttt{reduce} : (\alpha \to \alpha \to \alpha) \to \alpha \to []\alpha \to \alpha$$

Then

$$\texttt{sum} = \texttt{reduce}\ (+)\ 0$$

Sadly, NumPy is a first-order programming model, and we cannot define an efficient general `reduce`.

Monoidal reduction

We can abstract the notion of summation into a higher-order function typically called `reduce`:

$$\texttt{reduce} : (\alpha \to \alpha \to \alpha) \to \alpha \to []\alpha \to \alpha$$

Then

$$\texttt{sum} = \texttt{reduce}\ (+)\ 0$$

Sadly, NumPy is a first-order programming model, and we cannot define an efficient general `reduce`.

But first order programming models have even bigger problems.

The problem with control flow

It is awkward to encode per-workitem control flow in a first-order parallel language.

The problem with control flow

It is awkward to encode per-workitem control flow in a first-order parallel language.

$$y[i] = \begin{cases} \sqrt{x[i]} & \text{if } 0 \leq x[i] \\ x[i] & \text{otherwise} \end{cases}$$

The problem with control flow

It is awkward to encode per-workitem control flow in a first-order parallel language.

$$y[i] = \begin{cases} \sqrt{x[i]} & \text{if } 0 \leq x[i] \\ x[i] & \text{otherwise} \end{cases}$$

```
def sqrt_when_pos_0(x):
    x = x.copy()
    for i in range(len(x)):
        if x[i] >= 0:
            x[i] = np.sqrt(x[i])
    return x
```

The problem with control flow

It is awkward to encode per-workitem control flow in a first-order parallel language.

$$y[i] = \begin{cases} \sqrt{x[i]} & \text{if } 0 \leq x[i] \\ x[i] & \text{otherwise} \end{cases}$$

```
def sqrt_when_pos_0(x):
    x = x.copy()
    for i in range(len(x)):
        if x[i] >= 0:
            x[i] = np.sqrt(x[i])
    return x
```

Runtime for $n = 10^6$: 14s

Using flags, filters, and scatters

```
def sqrt_when_pos_1(x):
    x = x.copy()
    x_nonneg = x >= 0
    x[x_nonneg] = np.sqrt(x[x_nonneg])
    return x
```

Using flags, filters, and scatters

```
def sqrt_when_pos_1(x):
    x = x.copy()
    x_nonneg = x >= 0
    x[x_nonneg] = np.sqrt(x[x_nonneg])
    return x
```

```
x           [    1,    2,    -3]
x_nonneg    [ True, True, False]
x[x_nonneg]                [1, 2]
```

Using flags, filters, and scatters

```
def sqrt_when_pos_1(x):
    x = x.copy()
    x_nonneg = x >= 0
    x[x_nonneg] = np.sqrt(x[x_nonneg])
    return x
```

```
x           [    1,    2,    -3]
x_nonneg    [ True, True, False]
x[x_nonneg]                [1, 2]
```

- Parallel filtering is not so simple (you'll see in a later lecture).
- Runtime for $n = 10^6$: 0.18s (vs 14s before!)

Arithmetic control flow

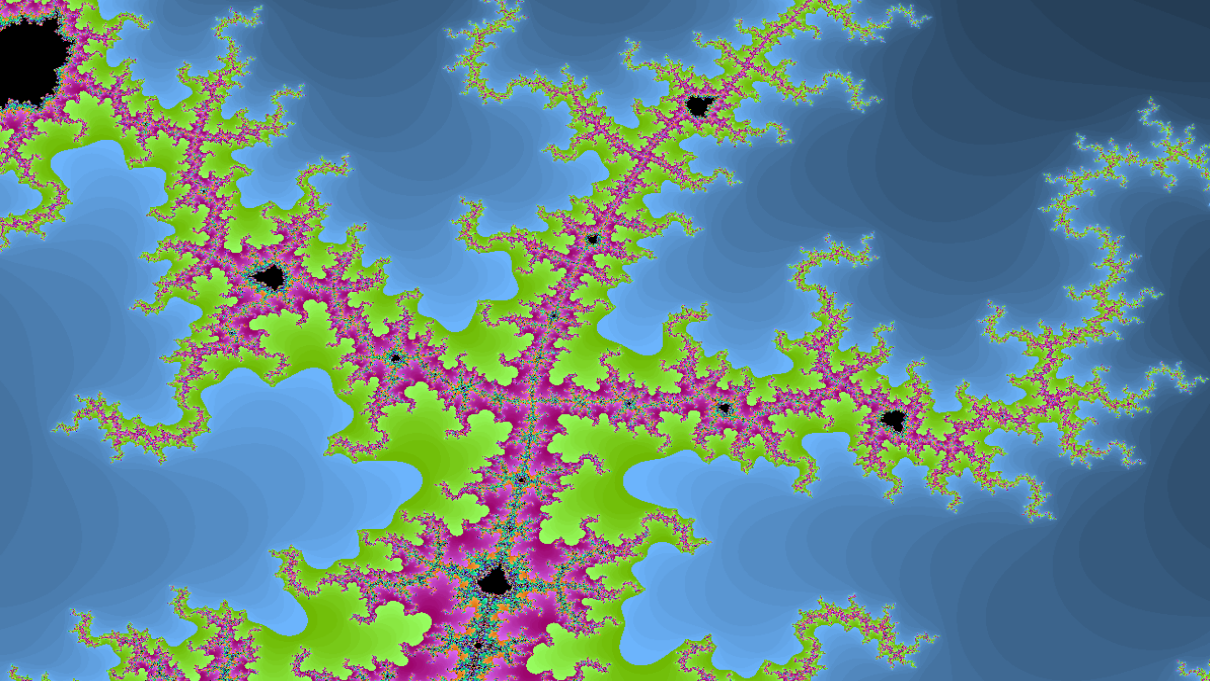
```
def sqrt_when_pos_2(x):
    x_nonneg = x >= 0
    x_neg = x < 0
    x_zero_when_neg = x * x_nonneg.astype(int)
    x_zero_when_nonneg = x * x_neg.astype(int)
    x_sqrt_or_zero = np.sqrt(x_zero_when_neg)
    return x_sqrt_or_zero + x_zero_when_nonneg
```

```
x                 [    1,         2,    -3]
x_nonneg          [ True,      True, False]
x_neg             [False,     False,  True]
x_zero_when_neg   [    1,         2,     0]
x_zero_when_nonneg[    0,         0,    -3]
x_sqrt_or_zero    [    1, 1.4142135,     0]
return            [    1, 1.4142135,    -3]
```

Runtime for $n = 10^6$: 0.10s.

It gets worse

That's for a simple conditional. What if we need a *loop*?



Computing the Mandelbrot Set

The root of those pretty visuals is calling this function (pseudocode) with a bunch of complex numbers:

```
def divergence(c, d):
  i = 0
  z = c
  while i < d and dot(z) < 4.0:
    z = c + z * z
    i = i + 1
  return i
```

Mandelbrot in NumPy[3]

```
def mandelbrot_numpy(c, d):
    output = np.zeros(c.shape)
    z = np.zeros(c.shape, np.complex32)
    for it in range(d):
        notdone =
          (z.real*z.real + z.imag*z.imag) < 4.0
        output[notdone] = it
        z[notdone] = z[notdone]**2 + c[notdone]
    return output
```

Problems

- Control flow obscured.
- Always runs for *maxiter* iterations.
- *Lots* of memory traffic.

[3] https://www.ibm.com/developerworks/community/blogs/jfp/entry/How_To_Compute_Mandelbrodt_Set_Quickly

The root of the problem

- NumPy is a *first-order parallel language*—all operations are parameterised over simple values, not functions.
- Easy way to design and implement a library, but inflexible.

Now we will look at a *higher-order parallel language*: **Futhark**.

Content and motivation

Array programming

Higher-order array programming

Futhark is a high-level language!

- Futhark is *not* a "GPU language"—it is a hardware-agnostic parallel language.
- However, we have written a Futhark *compiler* that can generate good GPU code.
- The compiler is *not* a "parallelising compiler". The parallelism is *explicitly given* by the programmer. The compiler's job is to figure out what to do with it. It's more of a *sequentialising compiler*.
- *Co-design* between compiler and language, inspired by hand-written code for target hardware.

Futhark at a Glance

- **Size-dependent types**

  An n by m integer matrix has type `[`n`][`m`]i32`.

  - In the full language, there is an entire type theory to deal with existentially quantified sizes.
  - For this lecture, we assume that the sizes are already known symbolically.

Futhark at a Glance

- **Size-dependent types**

  An *n* by *m* integer matrix has type `[`*n*`][`*m*`]i32`.
  - In the full language, there is an entire type theory to deal with existentially quantified sizes.
  - For this lecture, we assume that the sizes are already known symbolically.

- **Nested Parallelism**

```
let add_two [n]     (a:  [n]i32):     [n]i32 = map (+2) a
let     sum [n]     (a:  [n]i32):        i32 = reduce (+) 0 a
let sumrows [n][m] (as: [n][m]i32): [n]i32 = map sum as
```

Futhark at a Glance

- **Size-dependent types**

  An n by m integer matrix has type `[`n`][`m`]i32`.
  - In the full language, there is an entire type theory to deal with existentially quantified sizes.
  - For this lecture, we assume that the sizes are already known symbolically.
- **Nested Parallelism**

```
let add_two [n]     (a:  [n]i32):     [n]i32 = map (+2) a
let     sum [n]     (a:  [n]i32):        i32 = reduce (+) 0 a
let sumrows [n][m] (as: [n][m]i32): [n]i32 = map sum as
```

- **Parallelism must be *regular*; this is not supported:**

```
let f [n] (a: [n]i32) =
  map (\x -> reduce (+) 0 (iota x)) a
```

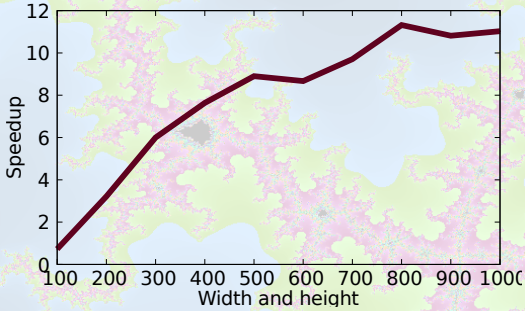
Mandelbrot in Futhark

```
let divergence (c: complex) (d: i32): i32 =
  let (_, i') =
    loop (z, i) = (c, 0)
    while i < d && dot(z) < 4.0 do
      (addComplex c (multComplex z z),
       i + 1)
  in i'

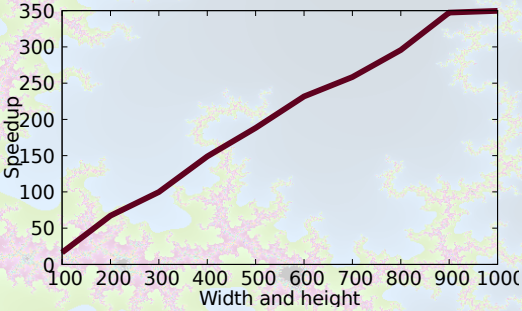
let mandelbrot [n][m] (css: [n][m]complex)
                      (d: i32) : [n][m]i32 =
  map (\cs -> map (\c -> divergence c d) cs) css
```

- Only one array written, at the end.
- `while` loop terminates when the element diverges.

Mandelbrot speedup on GPU vs sequential implementation in C



NumPy-style



Futhark-style

The vectorised style can sacrifice a lot of potential performance.

On Futhark and programming models

Full disclosure

Futhark was designed and implemented by colleges of mine at DIKU, Univ. Cph.

- Futhark is a **particular** and **restricted** implementation of the ideas to parallelism.
- But sadly it is one of the **few** high-level parallel languages that usually run **fast** (mostly).
- **Be careful not to conflate Futhark's implementation limitations and other quirks with the fundamental principles of data parallel programming.**

Summary

- Parallel programming is a physical necessity.
- Good language models combined with good compilers can be used to make hardware more efficient.
- Deterministic data parallel programming is demonstrably both accessible to humans and efficient to execute.
- A first-order model is insufficient.
- Sheer parallelism is not enough to make it fast.
- The principles are beautiful, but extant tools and languages often fall short.

Go install and try Futhark: `https://futhark-lang.org`

Possible exercises: `https://github.com/diku-dk/dpp-e2021-pub/blob/master/bootstrap-exercises.md`