Deterministic parallel programming and data parallelism

Troels Henriksen (athas@sigkill.dk)

DIKU
University of Copenhagen

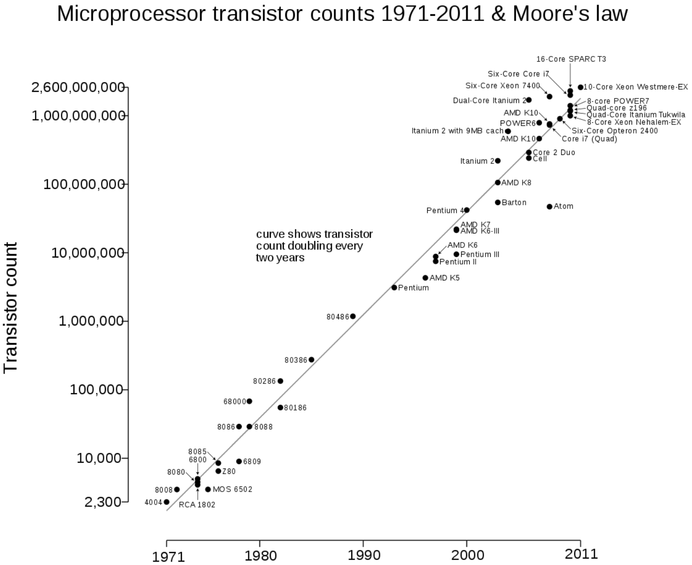
Our predicament

The need for a programming model

Array programming with NumPy

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.

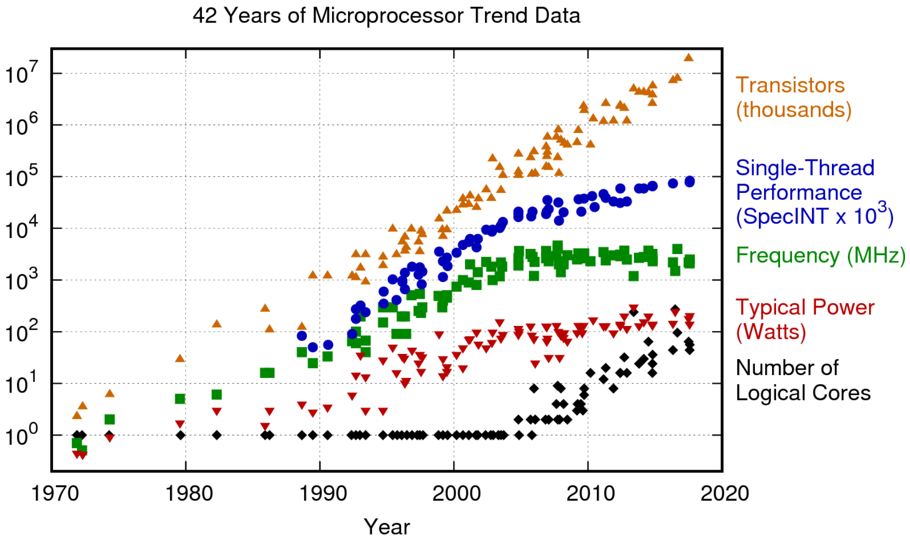
Moore's Law

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.
  - ...meaning more instructions per second!

CPU trends over 40 years



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

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.
- Even Moore's Law will eventually 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.

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.

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

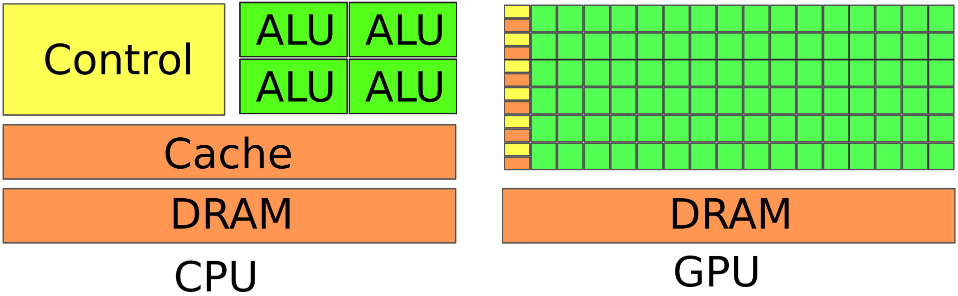
Simultaneously performing *the same* operation on different pieces of the same data.
Almost always deterministic.

The style of OpenMP you've been taught is data parallelism.

Case study: Graphics Processing Units (GPUs)

- Originally arose as **specialised graphics processors** in the 90s.
- Eventually became **highly efficient massively parallel processors**.
- Their efficiency is basically the reason for the current AI revolution.

GPUs vs CPUs



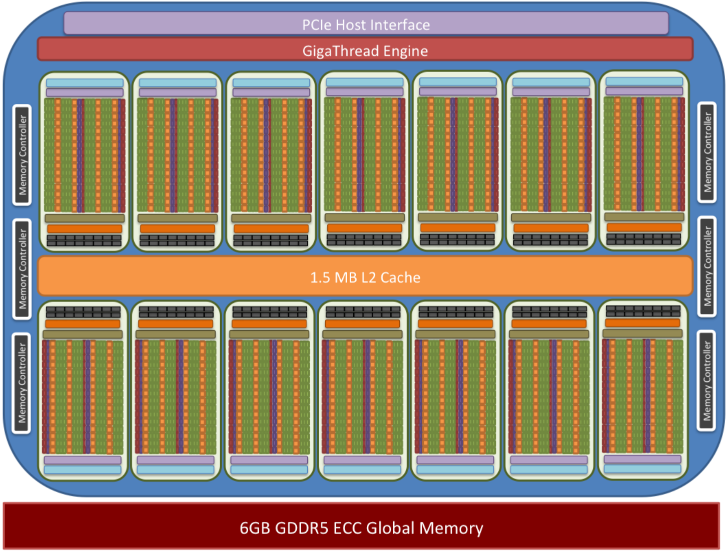
- GPUs have *thousands* of simple cores and taking full advantage of their compute power requires *tens of thousands* of threads.
- GPU threads are very *restricted* in what they can do: no stack, no allocation, limited control flow, etc.
- Potential *very high performance* and *lower power usage* compared to CPUs, but programming them is *hard*

Massive parallelism is currently a special case but becoming increasingly common.

Execution Model

- The GPU bundles threads in groups of 32, called *warps*. These are the unit of scheduling
- Using *oversubscription* (many more threads that can run simultaneously) and *zero-overhead hardware scheduling*, the GPU can aggressively *hide latency*
- Following illustrations from `https://www.olcf.ornl.gov/for-users/system-user-guides/titan/nvidia-k20x-gpus/`
  Older K20 chip (2012), but modern architectures are very similar.

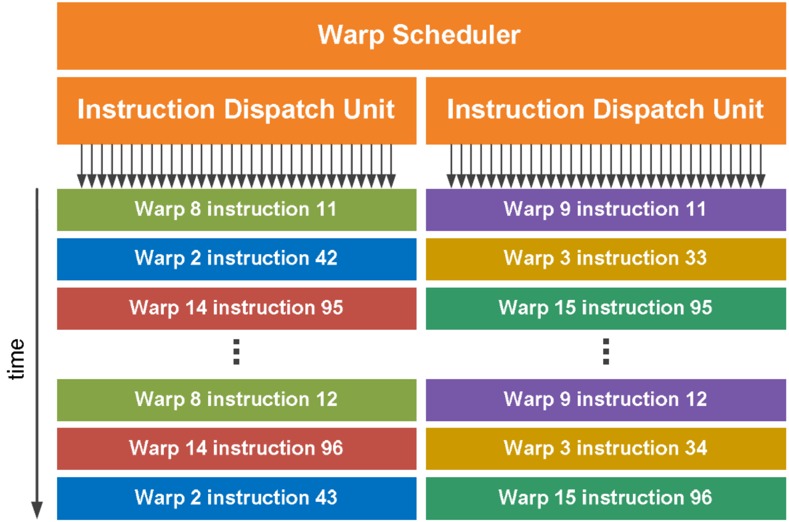
K20 GPU layout—an assembly of 14 *streaming multiprocessors*



Streaming multiprocessor layout

C single precision/integer CUDA core
DP double precision FP unit
L/S memory load/store unit
SF special function unit

Warp scheduling



Two guiding quotes

When we had no computers, we had no programming problem either. When we had a few computers, we had a mild programming problem. Confronted with machines a million times as powerful, we are faced with a gigantic programming problem.

—Edsger W. Dijkstra (EWD963, 1986)

Two guiding quotes

When we had no computers, we had no programming problem either. When we had a few computers, we had a mild programming problem. Confronted with machines a million times as powerful, we are faced with a gigantic programming problem.

—Edsger W. Dijkstra (EWD963, 1986)

The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague.

—Edsger W. Dijkstra (EWD340, 1972)

Human brains simply cannot reason about concurrency on a massive scale

- We need a programming model with *sequential* semantics, but that can be *executed* in parallel.
- It must be *portable*, because hardware continues to change.
- It must support *modular* programming, so we can write reusable components.

Questions that researchers need to answer

What we want

Hardware that maximises useful work per transistor and watt.

What we need

High-level human-friendly programming models with *sequential semantics* that can be efficiently compiled to the efficient hardware.

What we don’t know yet

What should such a programming model look like? How is it mapped to hardware? Which interesting transformations are possible on programs within such a model?

Our predicament

The need for a programming model

Array programming with NumPy

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 $\oplus : \alpha \rightarrow \alpha \rightarrow \alpha$ 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 $\oplus : \alpha \rightarrow \alpha \rightarrow \alpha$ 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*

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

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

```python
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.
- Runtime for $n = 10^6$: 0.18s (vs 14s before!)

Arithmetic control flow

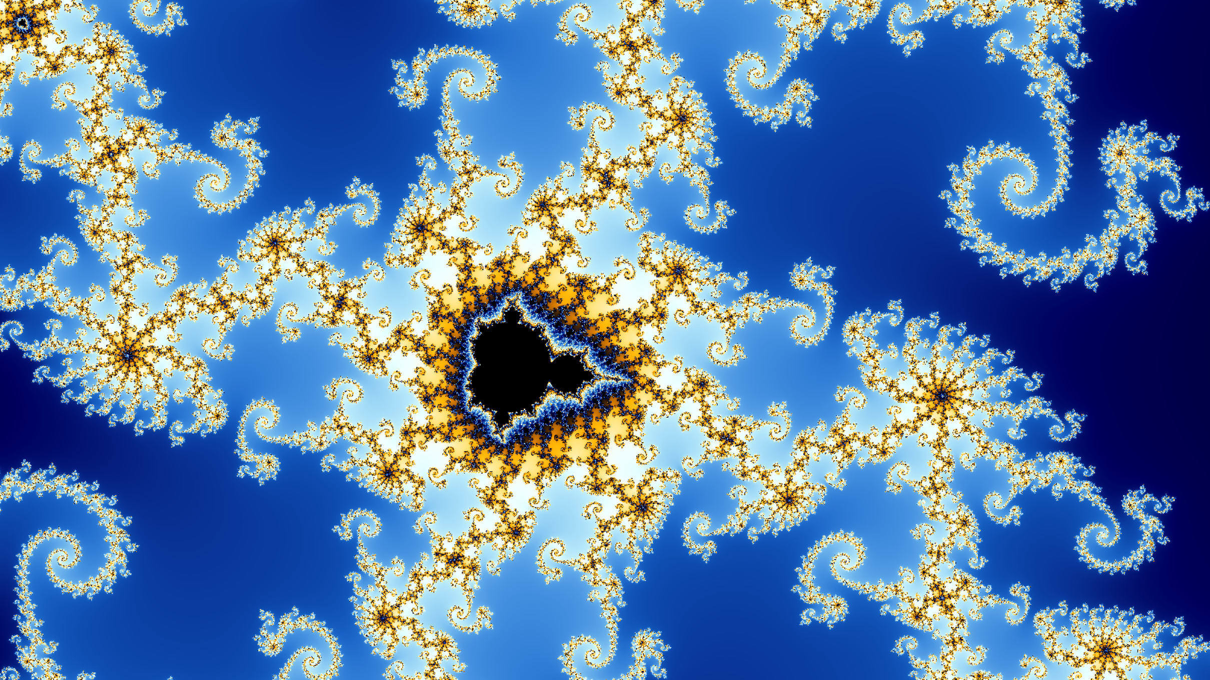
```
def sqrt_when_pos_2(x):
    x_nonneg = x >= 0
    x_neg = x < 0
    x_zero_when_neg = x * x_nonneg.astype(int)
    x_zero_when_nonneg = x * x_neg.astype(int)
    x_sqrt_or_zero = np.sqrt(x_zero_when_neg)
    return x_sqrt_or_zero + x_zero_when_nonneg
```

```
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

Rendered by calling this function 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
```

- Viewing the complex number field as a 2D coordinate system, each pixel position (x, y) corresponds to a complex number.
- The integer return value of `divergence` can be turned into a colour.

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**.

Our predicament

The need for a programming model

Array programming with NumPy

Higher-order array programming

Futhark is a high-level language!

- Futhark is maintained by a team at DIKU, including Cosmin Oancea, Martin Elsman, Philip Munksgaard, Robert Schenck, and myself.
- 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.
- **Not general purpose**—only used for the computational core of programs.

Futhark at a Glance

- **Size-dependent types**

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

  - Compiler decides in-memory layout (e.g. row- or column major) based on how array is used.

Futhark at a Glance

- **Size-dependent types**

  An *n* by *m* integer matrix has type `[n][m]i32`.
  - Compiler decides in-memory layout (e.g. row- or column major) based on how array is used.
- **Nested Parallelism**

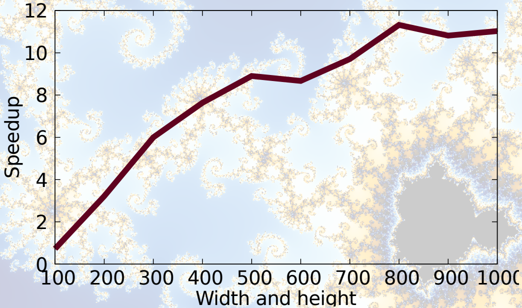
```
def add_two [n]    (a:  [n]i32):    [n]i32 = map (+2) a
def     sum [n]    (a:  [n]i32):       i32 = reduce (+) 0 a
def sumrows [n][m] (as: [n][m]i32): [n]i32 = map sum as
```

Mandelbrot in Futhark

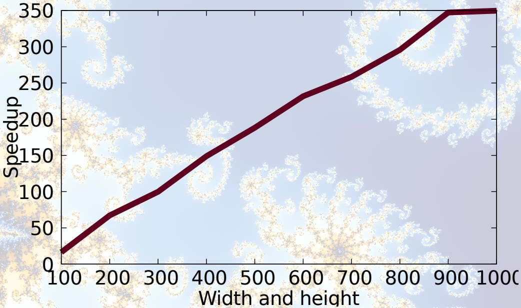
```
def divergence (c: complex) (d: i32): i32 =
  let (_, i') =
    loop (z, i) = (c, 0)
    while i < d && dot(z) < 4.0 do
      (addComplex c (multComplex z z),
       i + 1)
  in i'

def 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.

Summary

- Parallel programming is a physical necessity.
- 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.

Courses if you like this stuff

- *Programming Massively Parallel Hardware*, taught by Cosmin Oancea in block 1
- *Data Parallel Programming*, taught by Cosmin Oancea and myself in block 2

Try out our research: `https://futhark-lang.org`