Futhark

A data-parallel pure functional programming language compiling to GPU

Troels Henriksen (athas@sigkill.dk)

Computer Science
University of Copenhagen

31. May 2016

ME

- PhD student at the Department of Computer Science at the University of Copenhagen (DIKU).
- Affiliated with the HIPERFIT research centre – *Functional High-Performance Computing for Financial IT*.
- I'm mostly interested in the *Functional High-Performance Computing* part.
- My research involves working on a high-level purely functional language, called Futhark, and its heavily optimising compiler. That language is what I'm here to talk about.

HIPERFIT.DK

Agenda

GPU programming

In which I hope to convince you that GPU programming is worthwhile, but also hard

Functional parallelism

We take a step back and look at functional representations of parallelism

Futhark to the rescue

My own language for high-performance functional programming

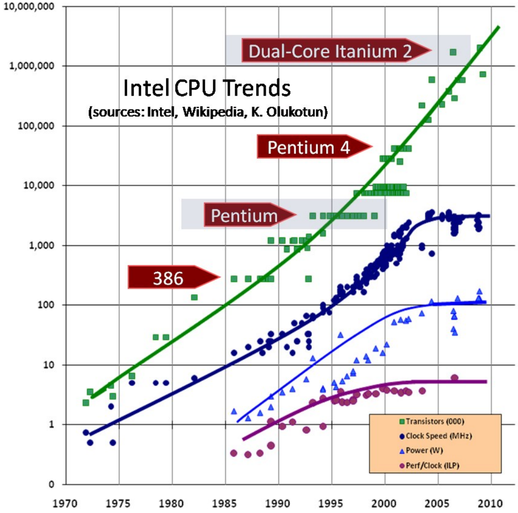
Part 0: GPU programming - why?

Transistors (thousands)

Clock Speed (MHz)

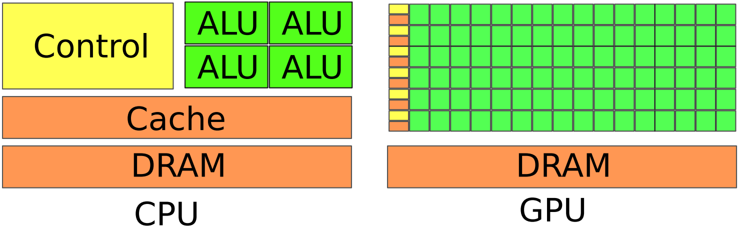
Power (W)

Perf/Clock (ILP)



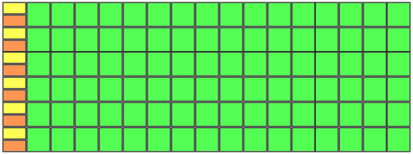
- Moore’s law still in effect, and will be for a while…
- … but we no longer get many increases in sequential performance.

CPU versus GPU architecture



- CPUs are all about *control*. The program can branch and behave dynamically, and the CPU has beefy caches and branch predictors to keep up.
- GPUs are all about *throughput*. The program has very simple control flow, and in return the transistor budget has been spent on computation.

The SIMT Programming Model



- GPUs are programmed using the SIMT model (*Single Instruction Multiple Thread*).
- Similar to SIMD (*Single Instruction Multiple Data*), but while SIMD has explicit vectors, we provide *sequential scalar per-thread* code in SIMT.

Each thread has its own registers, but they all execute the same instructions at the same time (i.e. they share their instruction pointer).

SIMT example

For example, to increment every element in an array a, we might use this code:

```
increment(a) {
  tid = get_thread_id();
  x = a[tid];
  a[tid] = x + 1;
}
```

- If `a` has `n` elements, we launch `n` threads, with `get_thread_id()` returning *i* for thread *i*.
- This is *data-parallel programming*: applying the same operation to different data.

Predicated Execution

If all threads share an instruction pointer, what about branches?

```
mapabs(a) {
  tid = get_thread_id();
  x = a[tid];
  if (x < 0) {
    a[tid] = -x;
  }
}
```

Masked Execution

Both branches are executed in all threads, but in those threads where the condition is false, a mask bit is set to treat the instructions inside the branch as no-ops.

When threads differ on which branch to take, this is called *branch divergence*, and can be a performance problem.

CUDA

I will be using NVIDIAs proprietary CUDA API in this presentation, as it is the most convenient for manual programming. OpenCL is an open standard, but more awkward to use.

- In CUDA, we program by writing a *kernel function*, and specifying a *grid* of threads.
- The grid consists of equally-sized *workgroups* of threads.
- Communication is only possible between threads within a single workgroup.
- The workgroup size is configurable, but at most 1024.

The grid is multi-dimensional (up to 3 dimensions), but this is mostly a programming convenience. I will be using a single-dimensional grid.

C: `map(+2, a)`

To start with, we will implement a function that increments every element of an array by two. In C:

```c
void increment_cpu(int num_elements, int *a) {
  for (int i = 0; i < num_elements; i++) {
    a[i] += 2;
  }
}
```

Note that every iteration of the loop is independent of the others. Hence we can trivially parallelise this by launching `num_elements` threads and asking thread *i* to execute iteration *i*.

CUDA: map(+2, a)

Kernel function:

```
__global__ void increment(int num_elements, int *a) {
  int thread_index = threadIdx.x + blockIdx.x * blockDim.x;

  if (thread_index < num_elements) {
    a[thread_index] += 2;
  }
}
```

Using the kernel function:

```
int *d_a;
cudaMalloc(&d_a, rows*columns*sizeof(int));

int group_size = 256; // arbitrary
int num_groups = divRoundingUp(num_elements, group_size);
increment<<<num_groups, group_size>>>(num_elements, d_a);
```

That’s not so bad. How fast is it?

GPU Performance

On a GeForce GTX 780 Ti, it processes a $5e6$ (five million) element array in $189us$. Is that good? How do we tell?

- Typically, GPU performance is limited by *memory bandwidth*, not *computation*. This is the case here - we load one integer, perform one computation, then write one integer. We are quite memory-bound.
- If we divide the total amount of memory traffic ($5e6 \cdot \texttt{sizeof(int)}\ \text{bytes} \cdot 2$) by the runtime, we get $197.1GiB/s$. This GPU has a rated peak memory bandwidth of around $300GiB/s$, so it's not so bad.
- The sequential CPU version runs in $6833us$ ($5.5GiB/s$), but it's not a fair comparison.

Flying Too Close to the Sun

That went pretty well, so let's try something more complicated: summing the rows of a two-dimensional array. We will give every thread a row, which it will then sum using a sequential loop

```
__global__ void sum_rows(int rows, int columns,
                         int *a, int *b) {
  int thread_index = threadIdx.x + blockIdx.x * blockDim.x;

  if (thread_index < rows) {
    int sum = 0;
    for (int i = 0; i < columns; i++) {
      sum += a[thread_index * columns + i];
    }
    b[thread_index] = sum;
  }
}
```

Easy! Should be bandwidth-bound again. Let's check the performance...

That Went Poorly!

The `sum_rows` program can process a 50000×100 array in $840us$, which corresponds to $22.4GiB/s$. This is **terrible**!
The reason is our memory access pattern – specifically, our loads are not *coalesced*.

Memory Coalescing

On NVIDIA hardware, all threads within each consecutive 32-thread *warp* should simultaneously access consecutive elements in memory to maximise memory bus usage.

- If neighboring threads access widely distant memory in the same clock cycle, the loads have to be *sequentialised*, instead of all fulfilled using one (wide) memory bus operation.
- The GTX 780 Ti has a bus width of 384 bits, so only using 32 bits per operation exploits less than a tenth of the bandwidth.

Transposing for Coalescing

Table: Current accesses - this is worst case behaviour!

Iteration	**Thread** 0	**Thread** 1	**Thread** 2	...
0	$A[0]$	$A[c]$	$A[2c]$	...
1	$A[1]$	$A[c+1]$	$A[2c+1]$	...
2	$A[2]$	$A[c+2]$	$A[2c+2]$	...

Table: These are the accesses we want

Iteration	**Thread** 0	**Thread** 1	**Thread** 2	...
0	$A[0]$	$A[1]$	$A[2]$	...
1	$A[c]$	$A[c+1]$	$A[c+2]$	...
2	$A[2c]$	$A[2c+1]$	$A[2c+2]$	...

This corresponds to accessing the array in a transposed or *column-major* manner.

Let Us Try Again

```
__global__ void sum_rows(int rows, int columns,
                         int *a, int *b) {
  int thread_index = threadIdx.x + blockIdx.x * blockDim.x;

  if (thread_index < rows) {
    int sum = 0;
    for (int i = 0; i < columns; i++) {
      sum += a[thread_index + i * rows];
    }
    b[thread_index] = sum;
  }
}
```

- Runs in 103*us* and accesses memory at 182.7*GiB*/*s*.
- Actually runs faster than our `map(+2, a)` (187*us*), because we don't have to store as many result elements.
- Works if a is stored in column-major form. We can accomplish this by explicltly transposing, which can be done efficiently (essentially at memory copy speed). The overhead of a transpose is much less than the cost of non-coalesced access.

This is not what you are used to!

On the CPUs, this transformation *kills* performance due to bad cache behaviour (from $11.4GiB/s$ to $4.0GiB/s$ in a single-threaded version):

```
void sum_rows_cpu(int rows, int columns,
                  int *a, int *b) {
  for (int j = 0; j < rows; j++) {
    int sum = 0;
    for (int i = 0; i < columns; i++) {
      sum += a[j * columns + i];
      // or slow: a[j + i * rows]
    }
    b[j] = sum;
  }
}
```

Access Patterns

On the CPU, you want memory traversals within a single thread to be *sequential*, on the GPU, you want them to be *strided*.

Insufficient Parallelism

We’ve used a 50000×100 array so far; let’s try some different sizes:

Insufficient Parallelism

We've used a 50000×100 array so far; let's try some different sizes:

Input size	Runtime	Effective bandwidth
50000×1000	$917us$	$203.3GiB/s$
5000×1000	$302us$	$61.7GiB/s$
500×1000	$267us$	$6.98GiB/s$
50×1000	$200us$	$0.93GiB/s$

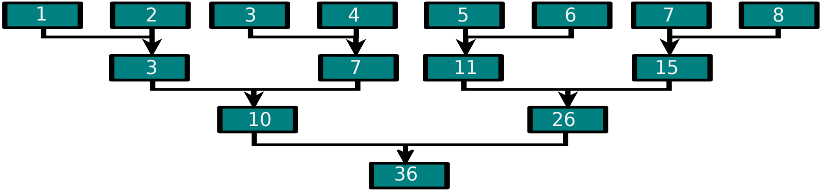
- The problem is that we only parallelise the *outer per-row loop*.
- Fewer than 50000 threads is too little to saturate the GPU, so much of it ends up sitting idle.
- We will need to parallelise the reduction of *each row* as well.
- This is where things start getting really uncomfortable.

Segmented reduction using one workgroup per row

A fully general solution is complex, so I will be lazy.

- One workgroup per row.
- Workgroup size equal to row size (i.e. number of columns).
- The threads in a workgroup cooperate in summing the row in parallel.
- **This limits row size to the maximum workgroup size (1024).**

The intra-workgroup algorithm is tree reduction:



Global versus local memory

GPUs have several kinds of memory - the two most important are *global* and *local* memory.

Global memory is plentiful (several GiB per GPU), but relatively slow and high-.

Local memory is sparse (typically 64*KiB* per multiprocessor, of which a GPU may have a few dozen), but very fast.

All threads in a workgroup execute on the same multiprocessor and can access the same shared local memory. We use workgroup-wide *barriers* to coordinate access. This is how threads in a workgroup can communicate.

Row summation kernel

```
__global__
void sum_rows_intra_workgroup_slow(int rows, int columns,
                                   int *a, int *b) {
  extern __shared__ int shared[];
  shared[threadIdx.x] =
    a[threadIdx.x + blockIdx.x * blockDim.x];

  for (int r = blockDim.x; r > 1; r /= 2) {
    if (threadIdx.x < r / 2) {
      int v = shared[threadIdx.x*2] +
              shared[threadIdx.x*2 + 1];
      __syncthreads();
      shared[threadIdx.x] = v;
    }
  }
  if (threadIdx.x == 0) { b[blockIdx.x] = shared[0]; }
}
```

Actually, that’s not all of it - the real kernel exploits that you don’t need barriers within each 32-thread *warp*, and uses a better shared memory access pattern – but it does not fit on a slide.

Performance

	Outer parallelism		All parallelism	
Input size	Runtime	Bandwidth	Runtime	Bandwidth
50000×1000	$917us$	$203.3GiB/s$	$2718us$	$68.6GiB/s$
5000×1000	$302us$	$61.7GiB/s$	$290us$	$63.4GiB/s$
500×1000	$267us$	$6.98GiB/s$	$45us$	$41.4GiB/s$
50×1000	$200us$	$0.93GiB/s$	$20us$	$9.3GiB/s$

Performance

	Outer parallelism		All parallelism	
Input size	Runtime	Bandwidth	Runtime	Bandwidth
50000×1000	$917us$	$203.3GiB/s$	$2718us$	$68.6GiB/s$
5000×1000	$302us$	$61.7GiB/s$	$290us$	$63.4GiB/s$
500×1000	$267us$	$6.98GiB/s$	$45us$	$41.4GiB/s$
50×1000	$200us$	$0.93GiB/s$	$20us$	$9.3GiB/s$

- The optimal implementation depends on the input size. An efficient program will need to have both, picking the best at runtime.
- **This is an important point:** exploiting inner parallelism has a *cost*, but sometimes that cost is worth paying.
- Writing efficient GPU code is already painful; writing multiple optimised versions for different input size characteristics transcends human capacity for suffering (i.e. *you will never want to do this*).

It goes on…

There are *many more difficulties* that I will not describe in detail. It is a bad situation.

It goes on…

There are *many more difficulties* that I will not describe in detail. It is a bad situation.

1. How can take advantage of the GPU without worrying about these low-level details?

It goes on...

There are *many more difficulties* that I will not describe in detail. It is a bad situation.

1. How can take advantage of the GPU without worrying about these low-level details? **Functional programming**.

It goes on...

There are *many more difficulties* that I will not describe in detail. It is a bad situation.

1. How can take advantage of the GPU without worrying about these low-level details? **Functional programming**.
2. How can we write small *reusable* components that we can combine into efficient GPU programs?

It goes on...

There are *many more difficulties* that I will not describe in detail. It is a bad situation.

1. How can take advantage of the GPU without worrying about these low-level details? **Functional programming**.
2. How can we write small *reusable* components that we can combine into efficient GPU programs? **An optimising compiler taking advantage of functional invariants.**

For example, how do we combine our CUDA `increment` function with `sum_rows`?

Functional Data-Parallel Programming

Two Kinds of Parallelism

- **Task parallelism** is the simultaneous execution of different functions across the same or different datasets.
- **Data parallelism** is the simultaneous execution of the same function across the elements of a dataset.

The humble `map` is the simplest example of a data-parallel operator:

$$\texttt{map}(f, [v_0, v_1, \ldots, v_{n-1}]) = [f(v_0), f(v_1), \ldots, f(v_{n-1})]$$

But we also have `reduce`, `scan`, `filter` and others. These are no longer user-written functions, but *built-in language constructs* with parallel execution semantics.

Parallel Code in an Imaginary Functional Language

And a function that sums an array of integers:

$$\begin{aligned}\texttt{sum} &: [\mathit{int}] \to \mathit{int}\\ \texttt{sum}(a) &= \texttt{reduce}(+, 0, a)\end{aligned}$$

And a function that sums the rows of an array:

$$\begin{aligned}\texttt{sumrows} &: [[\mathit{int}]] \to \mathit{int}\\ \texttt{sumrows}(a) &= \texttt{map}(\texttt{sum}, a)\end{aligned}$$

And a function that increments every element by two:

$$\begin{aligned}\texttt{increment} &: [[\mathit{int}]] \to [[\mathit{int}]]\\ \texttt{increment}(a) &= \texttt{map}(\lambda r.\texttt{map}(+2, r), a)\end{aligned}$$

Loop Fusion

Let’s say we wish to first call `increment`, then `sumrows` (with some matrix *a*):

`sumrows(increment(`*a*`))`

- A naive compiler would first run `increment`, producing an entire matrix in memory, then pass it to `sumrows`.
- This problem is bandwidth-bound, so unnecessary memory traffic will impact our performance.
- Avoiding unnecessary intermediate structures is known as *deforestation*, and is a well known technique for functional compilers.
- It is easy to implement for a data-parallel language as *loop fusion*.

An Example of a Fusion Rule

The `map`-`map` Fusion Rule

The expression

$$\texttt{map}(f, \texttt{map}(g, a))$$

is *always* equivalent to

$$\texttt{map}(f \circ g, a)$$

- This is an extremely powerful property that is only true in the absence of side effects.
- Fusion is *the* core optimisation that permits the efficient decomposition of a data-parallel program.
- A full fusion engine has much more awkward-looking rules (`zip`/`unzip` causes lots of bookkeeping), but safety is guaranteed.

A Fusion Example

$$\begin{array}{rr}
\texttt{sumrows}(\texttt{increment}(a)) = & \text{(Initial expression)}\\
\texttt{map}(\texttt{sum}, \texttt{increment}(a)) = & \text{(Inline }\texttt{sumrows}\text{)}\\
\texttt{map}(\texttt{sum}, \texttt{map}(\lambda r.\texttt{map}(+2, r), a)) = & \text{(Inline }\texttt{increment}\text{)}\\
\texttt{map}(\texttt{sum} \circ (\lambda r.\texttt{map}(+2, r)), a) = & \text{(Apply }\texttt{map}\text{-}\texttt{map}\text{ fusion)}\\
\texttt{map}(\lambda r.\texttt{sum}(\texttt{map}(+2, r)), a) = & \text{(Apply composition)}
\end{array}$$

- We have avoided the temporary matrix, but the composition of `sum` and the `map` also holds an opportunity for fusion – specifically, `reduce`-`map` fusion.
- Will not cover in detail, but a `reduce` can efficiently apply a function to each input element before engaging in the actual reduction operation.
- Important to remember: a `map` going into a `reduce` is an efficient pattern.

So Functional Programming Solves the Problem and We Can Go Home?

Functional programming *is* a very serious contender, but current mainstream languages have significant issues.

So Functional Programming Solves the Problem and We Can Go Home?

Functional programming *is* a very serious contender, but current mainstream languages have significant issues.

Much too expressive

GPUs are bad at control flow, and most functional languages are *all about* advanced control flow.

So Functional Programming Solves the Problem and We Can Go Home?

Functional programming *is* a very serious contender, but current mainstream languages have significant issues.

Much too expressive
GPUs are bad at control flow, and most functional languages are *all about* advanced control flow.

Recursive data types
Linked lists are a no-go, because you cannot efficiently launch a thread for every element. Getting to the end of an n-element list takes $O(n)$ work by itself! We need arrays with (close to) random access.

So Functional Programming Solves the Problem and We Can Go Home?

Functional programming *is* a very serious contender, but current mainstream languages have significant issues.

Much too expressive

GPUs are bad at control flow, and most functional languages are *all about* advanced control flow.

Recursive data types

Linked lists are a no-go, because you cannot efficiently launch a thread for every element. Getting to the end of an n-element list takes $O(n)$ work by itself! We need arrays with (close to) random access.

Laziness

Lazy evaluation is basically control flow combined with shared state. Not gonna fly.

And the Most Important Reason

Even with a suitable functional language, it takes serious effort to write an optimising compiler that can compete with hand-written code.

- Restricting the language too much makes the compiler easier to write, but fewer interesting programs can be written. Examples: Thrust for C++, Accelerate for Haskell.
- If the language is too powerful, the compiler becomes impossible to write. Example: NESL (or Haskell, or SML, or Scala, or F#...)

How We Hope To Solve This

Our idea is to look at the kind of algorithms that people currently run on GPUs, and design a language that can express those.

In: Nested parallelism, explicit indexing, multidimensional arrays, arrays of tuples.

Out: Sum types, higher-order functions, unconstrained side effects, recursion(!).

We came up with a data-parallel array language called *Futhark*.

The Futhark Programming Language

Partly an intermediate language for DSLs, partly a vehicle for compiler optimisation studies, partly a language you can use directly.

- Supports nested, regular data-parallelism.
- Purely functional, but has support for efficient sequential code as well.
- Mostly targets GPU execution, but programming model suited for multicore too. Probably does not scale to clusters (no message passing or explicitly asynchronous parallelism).
- Very aggressive optimising compiler and OpenCL code generator.

Futhark at a Glance

Simple eagerly evaluated pure functional language with data-parallel looping constructs. Syntax is a combination of C, SML, and Haskell.

Data-parallel loops:

```
fun [int] addTwo([int] a) = map(+2, a)
fun int sum([int] a) = reduce(+, 0, a)
fun [int] sumrows([[int]] as) = map(sum, a)
```

Sequential loops:

```
fun int main(int n) =
  loop (x = 1) = for i < n do
    x * (i + 1)
  in x
```

Monomorphic first-order types:

Makes it harder to write powerful abstractions, but OK for optimisation.

Easy to Run

It is simple to run a Futhark program for testing.

```
fun int main(int n) = reduce(*, 1, map(1+, iota(n)))
```

Put this in `fact.fut` and compile:

```
$ futhark-c fact.fut
```

Now we have a program `fact`.

```
$ echo 10 | ./fact
3628800i32
```

Parallelisation is as easy as using `futhark-opencl` instead of `futhark-c`.

A More Complex Example

Let us write a gaussian blurr program. We assume that we are given an image represented as a value of type `[[[u8,3],cols],rows]`. We wish to split this into three arrays, one for each color channel.

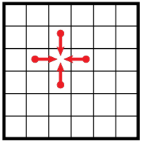
```
fun ([[f32,cols],rows],
     [[f32,cols],rows],
     [[f32,cols],rows])
  splitIntoChannels([[[u8,3],cols],rows] image) =
    unzip(map(fn [(f32,f32,f32),cols] ([[u8,3],cols] row) =>
                map(fn (f32,f32,f32) ([u8,3] pixel) =>
                      (f32(pixel[0]) / 255f32,
                       f32(pixel[1]) / 255f32,
                       f32(pixel[2]) / 255f32),
                    row),
                image))
```

Recombining the channels into one array

```
fun [[[u8,3],cols],rows]
  combineChannels([[f32,cols],rows] rs,
                  [[f32,cols],rows] gs,
                  [[f32,cols],rows] bs) =
    zipWith(fn [[u8,3],cols] ([f32,cols] rs_row,
                              [f32,cols] gs_row,
                              [f32,cols] bs_row) =>
              zipWith(fn [u8,3] (f32 r, f32 g, f32 b) =>
                        [u8(r * 255f32),
                         u8(g * 255f32),
                         u8(b * 255f32)],
                      rs_row, gs_row, bs_row),
              rs, gs, bs)
```

The Stencil Function

A *stencil* is a an operation where each array element is recomputed based on its neighbors.



```
fun f32 newValue([[f32,cols],rows] img, int row, int col) =
  unsafe
  let sum =
    img[row-1,col-1] + img[row-1,col] + img[row-1,col+1] +
    img[row,  col-1] + img[row,  col] + img[row,  col+1] +
    img[row+1,col-1] + img[row+1,col] + img[row+1,col+1]
  in sum / 9f32
```

Compute the average value of the pixel itself plus each of its eight neighbors. newValue(img, row, col) computes the new value for the pixel at position (row, col) in img.

The Full Stencil

We only call newValue on the inside, not the edges.

```
fun [[f32,cols],rows]
  blurChannel([[f32,cols],rows] channel) =
    map(fn [f32,cols] (int row) =>
          map(fn f32 (int col) =>
                if row > 0 && row < rows-1 &&
                   col > 0 && col < cols-1
                then newValue(channel, row, col)
                else channel[row,col],
              iota(cols)),
          iota(rows))
```

Putting It All Together

The `main` function accepts an image and a number of times to apply the gaussian blur.

```
fun [[[u8,3],cols],rows]
   main(int iterations, [[[u8,3],cols],rows] image) =
    let (rs, gs, bs) = splitIntoChannels(image)
    loop ((rs, gs, bs)) = for i < iterations do
      let rs = blurChannel(rs)
      let gs = blurChannel(gs)
      let bs = blurChannel(bs)
      in (rs, gs, bs)
    in combineChannels(rs, gs, bs)
```

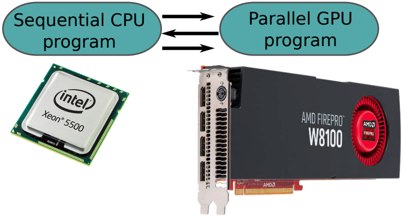
Testing it

```
$ (echo 100; futhark-dataset -g ’[[[i8,3],1000],1000]’) > input
$ futhark-c blur.fut -o blur-cpu
$ futhark-opencl blur.fut -o blur-gpu
$ ./blur-cpu -t /dev/stderr < input > /dev/null
1761245
$ ./blur-gpu -t /dev/stderr < input > /dev/null
43790
```

- 40× speedup over sequential CPU code - not bad.
- Standalone Futhark programs only useful for testing - let’s see how we can invoke the Futhark-generated code from a full application.

The CPU-GPU division

- The CPU uploads code and data to the GPU, queues kernel launches, and copies back results.
- **Observation:** the CPU code is all management and bookkeeping and does not need to be particularly fast.

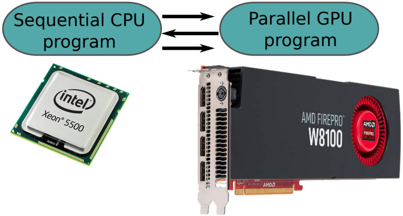


Sequential CPU program

Parallel GPU program

The CPU-GPU division

- The CPU uploads code and data to the GPU, queues kernel launches, and copies back results.
- **Observation:** the CPU code is all management and bookkeeping and does not need to be particularly fast.



How Futhark Becomes Useful

We can generate the CPU code in whichever language the rest of the user’s application is written in. This presents a convenient and conventional API, hiding the fact that GPU calls are happening underneath.

Compiling Futhark to Python+PyOpenCL

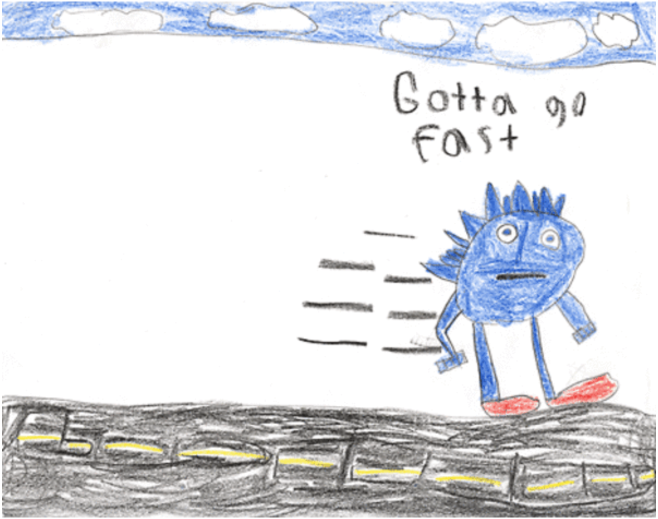
```
$ futhark-pyopencl blur.fut
```

This creates a Python module `blur.py` which we can use as follows:

```
import blur
import numpy
from scipy import misc
import argparse

def main(infile, outfile, iterations):
    b = blur.blur()
    img = misc.imread(infile, mode=’RGB’)
    (height, width, channels) = img.shape
    blurred = b.main(iterations, img)
    misc.imsave(outfile, blurred.get().astype(numpy.uint8))
```

Add some command line flag parsing and we have a nice little GPU-accelerated image blurring program.

The Spirit of Futhark - Original



The Spirit of Futhark - 1 Iteration

Gotta go
fast

```
$ python blur-png.py gottagofast.png \
--output-file gottagofast-1.png
```

The Spirit of Futhark - 100 Iterations

```
$ python blur-png.py gottagofast.png \
--output-file gottagofast-100.png --iterations 100
```

Other Nice Visualisations

These are made by having a Futhark program generate pixel data which is then fed to the Pygame library.

Performance

This is where you should stop trusting me!

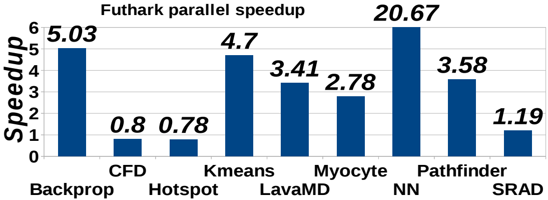
Performance

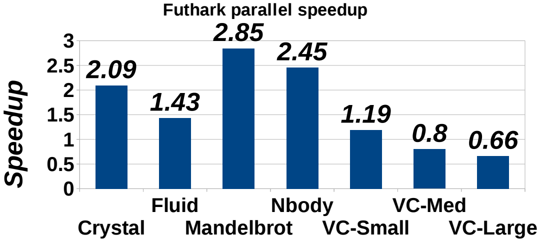
This is where you should stop trusting me!

- No good objective criterion for whether a language is “fast”.
- Best practice is to take benchmark programs written in other languages, port or re-implement them, and see how they behave.

Performance

Futhark compared to Accelerate and hand-written OpenCL from the Rodinia benchmark suite (higher is better).





Conclusions

- Simple functional language, yet expressive enough for nontrivial applications.
- Can be integrated with existing languages and applications.
- Performance is okay.

Questions?

Website	`http://futhark-lang.org`
Code	`https://github.com/HIPERFIT/futhark`
Benchmarks	`https://github.com/HIPERFIT/futhark-benchmarks`

HIPERFIT

Appendices

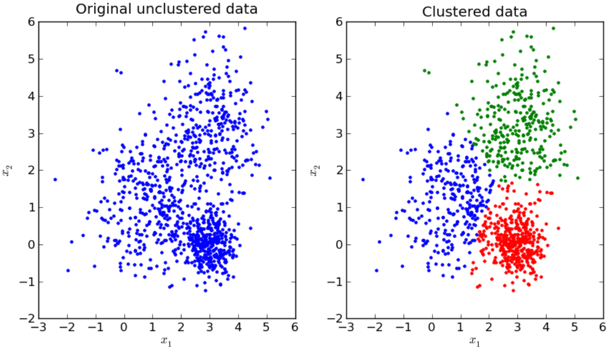
These will probably not be part of the presentation.

Case Study: *k*-means Clustering

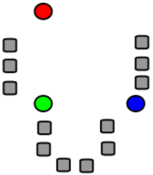
The Problem

We are given n points in some d-dimensional space, which we must partition into k disjoint sets, such that we minimise the inter-cluster sum of squares (the distance from every point in a cluster to the centre of the cluster).

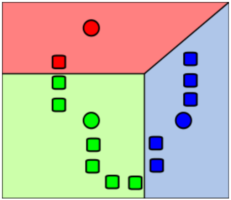
Example with $d = 2, k = 3, n = \textit{more than I can count}$:



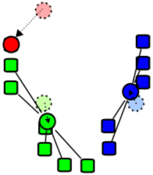
The Solution (from Wikipedia)



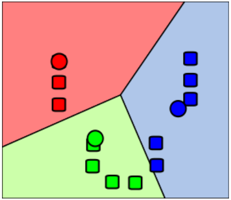
(1) *k* initial "means" (here $k = 3$) are randomly generated within the data domain.



(2) *k* clusters are created by associating every observation with the nearest mean.

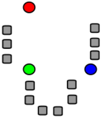


(3) The centroid of each of the *k* clusters becomes the new mean.



(4) Steps (2) and (3) are repeated until convergence has been reached.

Step (1) in Futhark

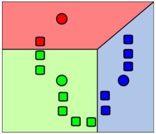


k initial ”means” (here $k = 3$) are randomly generated within the data domain.

`points` is the array of points - it is of type `[[f32,d],n]`, i.e. an `n` by `d` array of 32-bit floating point values.
We assign the first `k` points as the initial (“random”) cluster centres:

```
let cluster_centres = map(fn [f32,d] (int i) =>
                            points[i],
                          iota(k))
```

Step (2) in Futhark



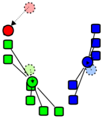
k clusters are created by associating every observation with the nearest mean.

```
-- Of type [int,n]
let new_membership =
  map(find_nearest_point(cluster_centres), points) in
```

Where

```
fun int find_nearest_point([[f32,d],k] cluster_centres,
                           [f32,d] pt) =
  let (i, _) = reduce(closest_point,
                      (0, euclid_dist_2(pt,cluster_centres[0]))
                      zip(iota(k),
                          map(euclid_dist_2(pt),
                              cluster_centres)))
  in i
```

Step (3) in Futhark



The centroid of each of the *k* clusters becomes the new mean.

This is the hard one.

```
let new_centres =
  centroids_of(k, points, new_membership)
```

Where

```
fun [[f32,d],k] centroids_of(int k,
                             [[f32,d],n] points,
                             [int,n] membership) =
  -- [int,k], the number of points per cluster
  let cluster_sizes = ...
  -- [[f32,d],k], the cluster centres
  let cluster_centres = ...
  in cluster_centres
```

Computing Cluster Sizes: the Ugly

A sequential loop:

```
loop (counts = replicate(k,0)) =
  for i < n do
    let cluster = membership[i]
    let counts[cluster] = counts[cluster] + 1
    in counts
```

This does what you think it does, and uses *uniqueness types* to ensure that the in-place array update is safe. This is what it looks like desugared:

```
loop (counts = replicate(k,0)) =
  for i < n do
    let cluster = membership[i]
    let new_counts =
      counts with [cluster] <- counts[cluster] + 1
    in new_counts
```

The type checker ensures that the old array `counts` is not used again.

Computing Cluster Sizes: the Bad

Use a parallel `map` to compute “increments”, and then a `reduce` of these increments.

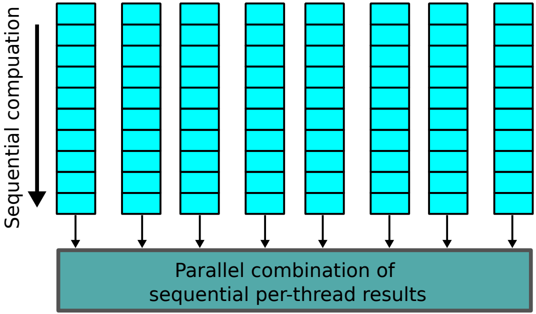
```
let increments =
  map(fn [int,k] (int cluster) =>
        let incr = replicate(k, 0)
        let incr[cluster] = 1
        incr,
      membership)

reduce(fn [int,k] ([int,k] x, [int,y]) =>
         zipWith(+, x, y),
       replicate(k, 0), increments)
```

This is parallel, but not work-efficient.

One Futhark Design Principle

Efficient Sequentialisation

The hardware is not infinitely parallel - ideally, we use an efficient sequential algorithm for chunks of the input, then use a parallel operation to combine the results of the sequential parts.



The optimal number of threads varies from case to case, so this should be abstracted from the programmer.

Computing cluster sizes: the Good

We use a Futhark language construct called a *reduction stream*.

```
let cluster_sizes =
  streamRed(fn [int,k] ([int,k] x, [int,k] y) =>
                 zipWith(+, x, y),

               fn [int,k] ([int,k] acc,
                           [int,chunksize] chunk) =>
                 loop (acc) = for i < chunksize do
                   let cluster = chunk[i]
                   let acc[cluster] = acc[cluster] + 1
                   in acc
                 in acc,

               replicate(k,0), membership) in
```

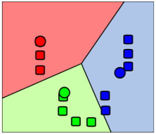
We specify a sequential fold function and a parallel reduction function. The compiler is able to exploit as much parallelism as is optimal on the hardware, and can use our sequential code inside each thread.

Back To Step (3)

Computing the actual cluster sums, now that we have `cluster_sizes`, is straightforwardly done with another stream:

```
let cluster_sums =
  streamRed(
    zipWith(zipWith(+)), -- matrix addition

    fn [[f32,d],k] ([[f32,d],k] acc,
                     [([f32,d], int),chunksize] chunk) =>
      loop (acc) = for i < chunksize do
        let (point, cluster) = chunk[i]
        let acc[cluster] =
          zipWith(+,
                   acc[cluster],
                   map(/cluster_sizes[cluster], point))
        in acc
      acc,
    replicate(k,replicate(d,0.0)),
    zip(points, membership))
```

Step (4) in Futhark



Steps (2) and (3) are repeated until convergence has been reached.

We iterate until we reach convergence (no points change cluster membership), or until we reach 500 iterations. This is done with a `while`-loop:

```
loop ((membership, cluster_centres, delta, i)) =
  while delta > 0 && i < 500 do
    let new_membership =
      map(find_nearest_point(cluster_centres), points)
    let new_centres =
      centroids_of(k, points, new_membership)
    let delta =
      reduce(+, 0,
             map(fn int (bool b) =>
                   if b then 0 else 1,
                 zipWith(==, membership, new_membership)))
    in (new_membership, new_centres, delta, i+1)
```

GPU Code Generation for the Reduction Streams

Reduction streams are easy-ish to map to GPU hardware, but there is still some work to do. The `cluster_sizes` stream will be broken up by the *kernel extractor* as:

```
-- produces an array of type [[int,k],num_threads]
let per_thread_results =
  chunkedMap(sequential code...)
-- combine the per-thread results
let cluster_sizes =
  reduce(zipWith(+), replicate(k, 0), per_thread_results)
```

The reduction with `zipWith(+)` is not great - the accumulator of a reduction should ideally be a scalar. The kernel extractor will recognise this pattern and perform a transformation called *Interchange Reduce With Inner Map* (IRWIM); moving the reduction inwards at a cost of a transposition.

After IRWIM

We transform

```
let cluster_sizes =
  reduce(zipWith(+), replicate(k, 0), per_thread_results)
```

and get

```
-- produces an array of type [[int,k],num_threads]
let per_thread_results =
  chunkedMap(sequential code...)
-- combine the per-thread results
let cluster_sizes =
  map(reduce(+, 0), transpose(per_thread_results))
```

- This interchange has changed the outer parallel dimension from being of size `num_threads` to being of size `k` (which is typically small).
- This is not a problem if we can translate `map(reduce(+, 0))` into a segmented reduction kernel (which it logically is).
- The Futhark compiler is smart enough to do this.

k-means Clustering: Performance

We compare the performance of the Futhark-implementation to a hand-written and hand-optimised OpenCL implementation from the Rodinia benchmark suite; the dataset is `kdd_cup`, where $n = 494025, k = 5, d = 35$.

k-means Clustering: Performance

We compare the performance of the Futhark-implementation to a hand-written and hand-optimised OpenCL implementation from the Rodinia benchmark suite; the dataset is `kdd_cup`, where $n = 494025, k = 5, d = 35$.

Rodinia:	0.864s
Futhark:	0.413s

This is OK, but we believe we can still do better.

1. We implement segmented reduction via segmented scan, which is nowhere near the most efficient implementation.
2. The Futhark compiler does a lot of unnecessary copying around of arrays.

Loop Interchange

Loop Interchange

Sometimes, to maximise the amount of parallelism we can exploit, we may need to perform *loop interchange*:

```
let bss = map(fn ([int,m]) (ps) =>
    let bs = loop (ws=ps) for i < n do
      let ws' = map(fn int (cs, w) =>
                     let d = reduce(+, 0, cs)
                     let e = d + w
                     let w' = 2 * e
                     in w',
                   css, ws)
      in ws'
    in bs,
  pss)
```

After interchange

We exploit that we can always interchange a parallel loop inwards, thus bringing the two parallel dimensions next to each other:

```
let bss = loop (wss=pss) for i < n do
  let wss' =
    map(fn [int,m] (css, ws) =>
      let ws' = map(fn int (cs, w) =>
                  let d  = reduce(+, 0, cs)
                  let e  = d + w
                  let w' = 2 * e
                  in w',
                css, ws)
      in ws',
    wss)
```

We can now continue to extract kernels from within the body of the loop.

Validity of Loop Interchange

Suppose that we have a this map-nest:

```
map(fn (x) =>
    loop (x'=x) for i < n do f(x'),
  xs)
```

Also suppose $\texttt{xs} = [x_0, x_1, \ldots, x_m]$, then the result of the map is

$$[f^n(x_0),\ f^n(x_1),\ \ldots,\ f^n(x_m)].$$

If we interchange the map inwards, then we get the following:

```
loop (xs'=xs) for i < n
  map(f, xs')
```

At the conclusion of iteration i, we have

$$xs' = [f^{i+1}(x_0),\ f^{i+1}(x_1),\ \ldots,\ f^{i+1}(x_m)].$$

At the conclusion of the last iteration $i = n - 1$, we have obtained the same result as the non-interchanged map. Note that the validity of the interchange does not depend on whether the for-loop contains a map itself.