Streaming data and garbage collection in Accelerate

Robert Clifton-Everest (robertce@cse.unsw.edu.au)
UNSW

GPUs

- Lots of raw computing power
  - This one: 2688 cores @ 867 MHz

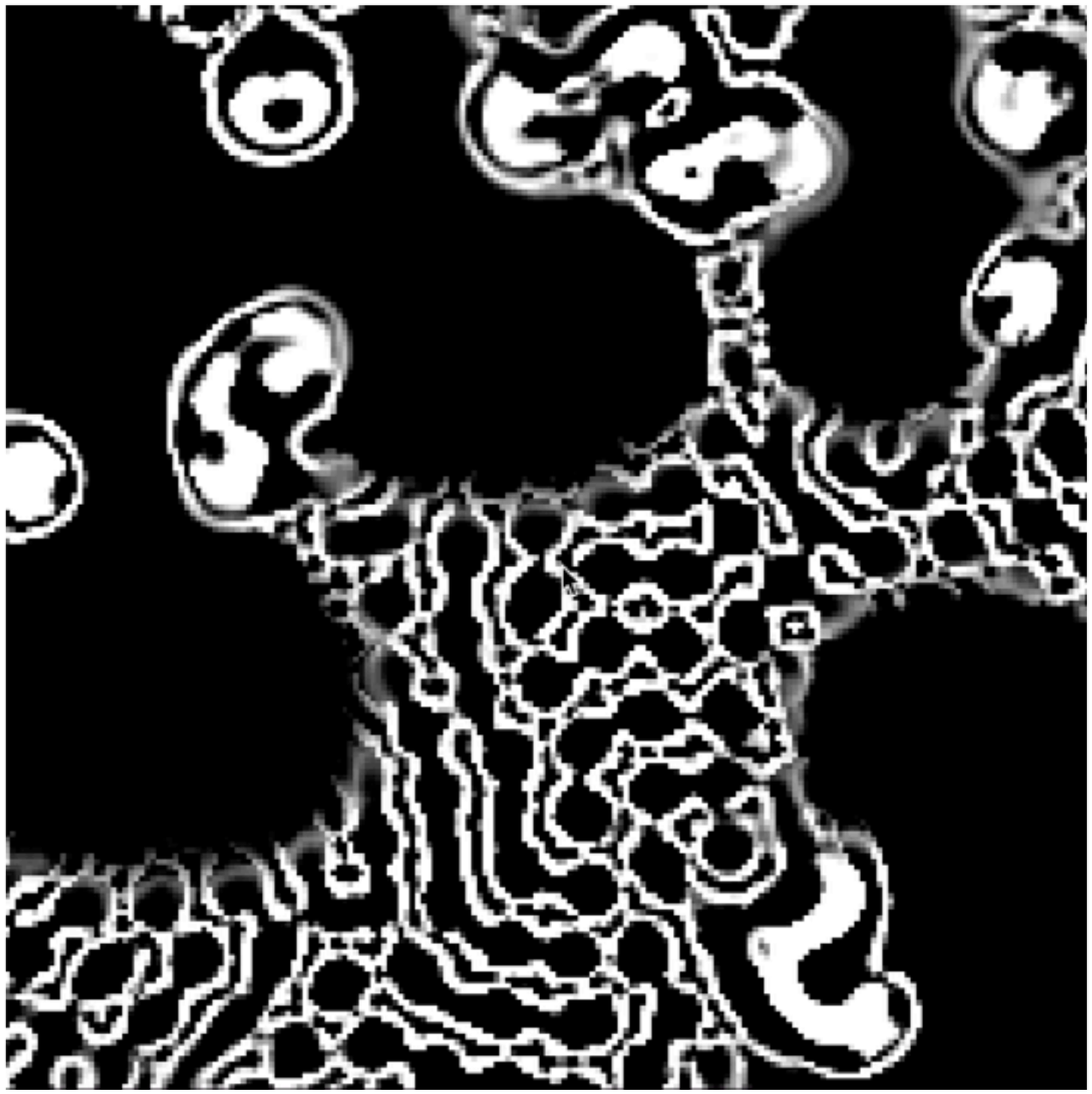
- Different hardware design
  - Limited instruction set
  - SIMD: Cores run the same program, but on different data

- How can we take advantage of this power?

With a high-level embedded language of course!

Accelerate

An embedded language for GPU programming



Accelerate

- A deep embedding

```
type Vector e = Array (Z:.Int) e
```

```
type Scalar e = Array Z e
```

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

```
fold :: (Exp e -> Exp e -> Exp e)
     -> Exp e
     -> Acc (Array (sh:.Int) e)
     -> Acc (Array sh e)
```

```
zipWith :: (Exp a -> Exp b -> Exp c)
        -> Acc (Array sh a)
        -> Acc (Array sh b)
        -> Acc (Array sh c)
```

Caenorhabditis elegans

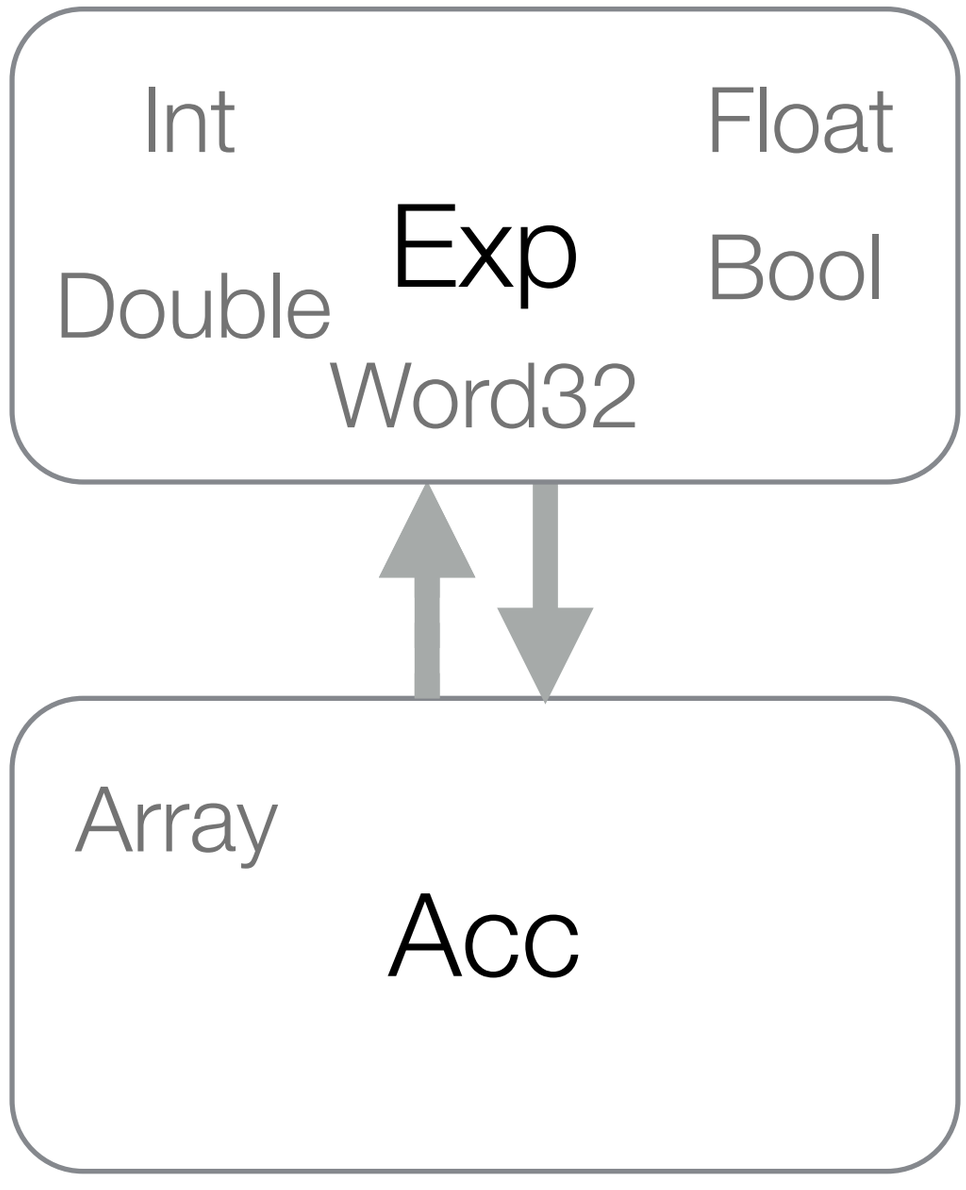
Worm processing

- Lots of data
  - GPU has relatively little memory
- Process a frame at a time?
  - Space issues reduced
  - What if frames are actually very small?

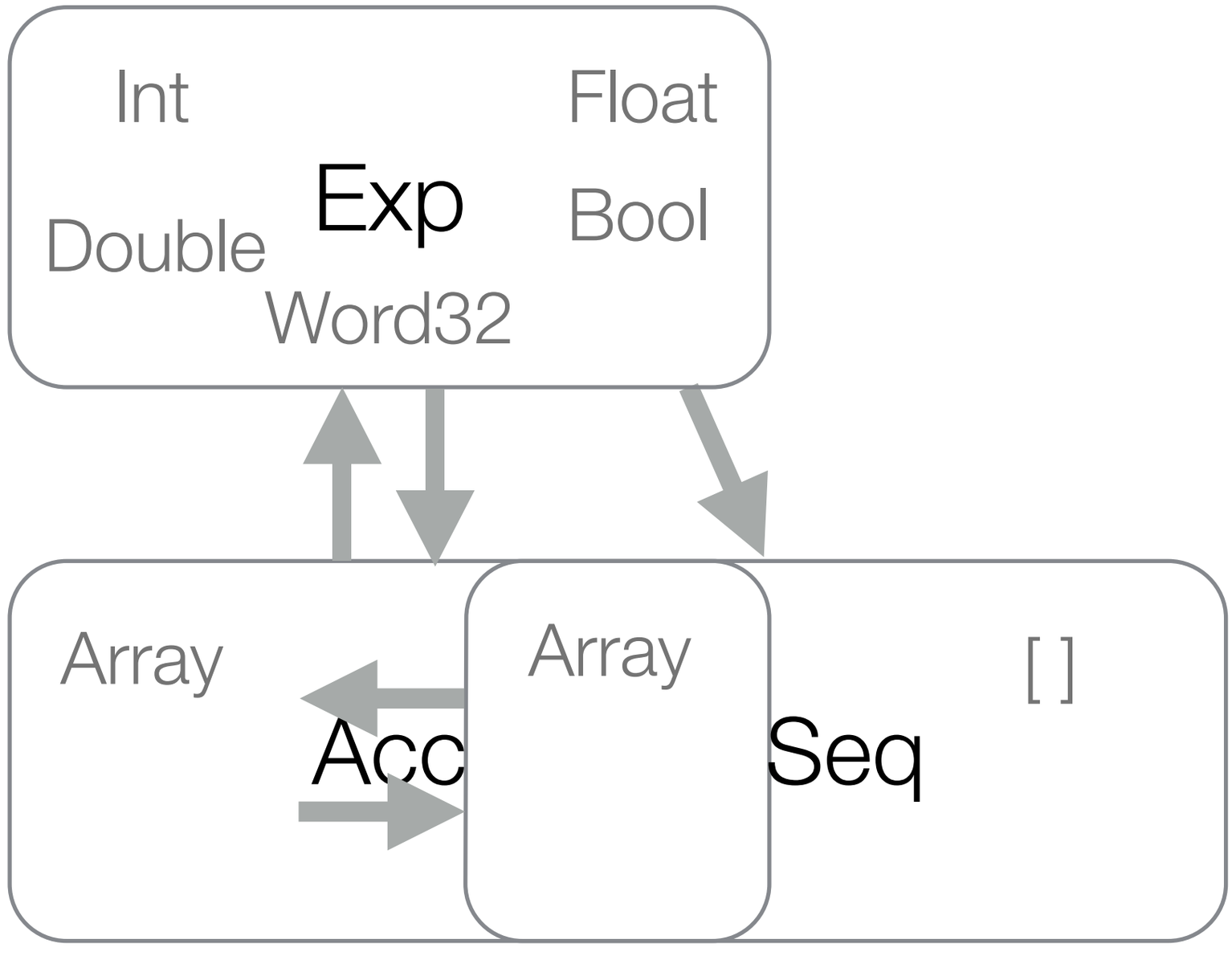
Sequences (streams)

- Sequences of arrays (or tuples of arrays)
- Can only be accessed linearly
- So map, fold and scan, but no permuting, indexing or constant time length
- Processed in chunks
  - Vectorisation
- Where do they fit in with the rest of the language?

Stratification



Sequences



Input and processing

```
streamIn :: Arrays a => [a] -> Seq [a]
```

Array or tuple of arrays (not a sequence)

```
toSeq :: Acc (Array (sh:.Int) e) -> Seq [Array sh e]
```

```
mapSeq :: (Acc a -> Acc b) -> Seq [a] -> Seq [b]
```

Output

```
streamOut :: Arrays a => Seq [a] -> [a]
```

```
fromSeq :: Seq [Array sh e] -> Seq (Vector sh, Vector e)
```

Elements not always the same size

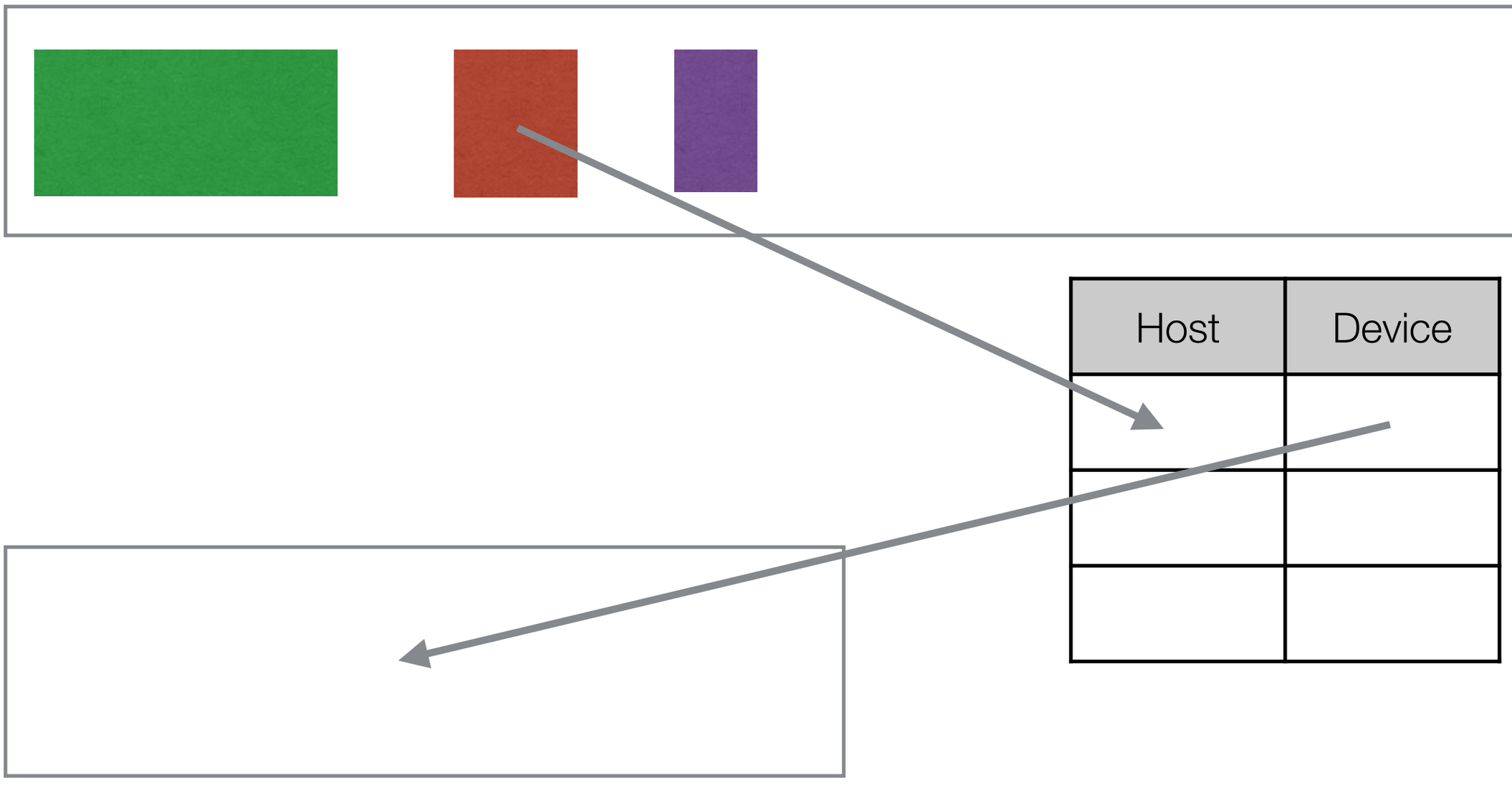
```
collect :: Arrays a => Seq a -> Acc a
```

How much of it works?

- Chunk-wise processing? Still ongoing.
- Out of core algorithms? Yes*

*See subsequent slides

Garbage Collection



The table

- Host keys
  - Stable names?

Stable names

```
data StableName a
```

```
makeStableName :: a -> IO (StableName a)
```

```
mkStableName a = mkStableName b ⟹ a = b
```

Do they make for a good key?

The table

- Host keys
  - Stable names? Not very stable.
  - Raw pointers? `data Array sh e = Array … Addr#`
    - Not unique, but it should be okay right?
- Removal
  - Weak pointers and finalizers.

Weak pointers

```
mkWeak :: k -> v -> Maybe (IO ()) -> IO (Weak v)
```

Establishes a weak pointer to k, with value v and a finalizer. that may run some arbitrary time in the future.

```
deRefWeak :: Weak v -> IO (Maybe v)
```

Dereferences a weak pointer. If the key is still alive, then `Just v` is probably returned (where v is the value of the weak pointer), otherwise `Nothing` is returned.

```
finalize :: Weak v -> IO ()
```

Causes the finalizer associated with a weak pointer to be run immediately. eventually

Dig deeper

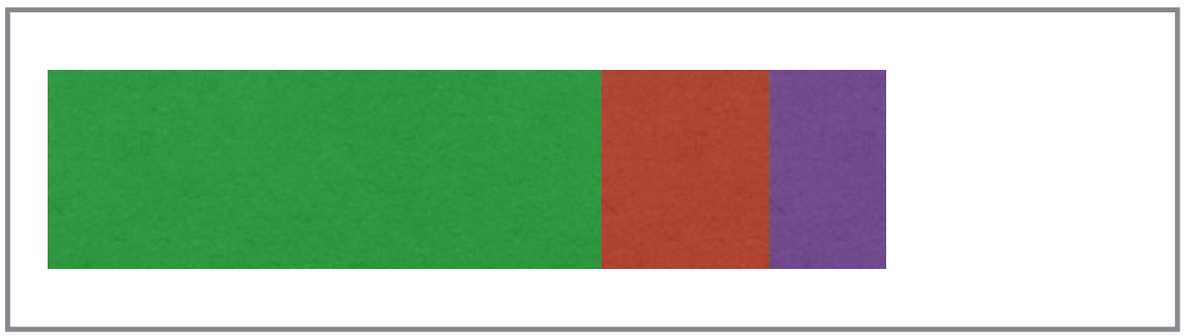
- Attach weak pointer to a primitive type.
  - Stops finalizers firing early
  - mkWeak#
  - Doesn't work for Addr# `data Array sh e = Array … Addr#`
  - Does work for MutVar# `data Array sh e = Array … MutVar# … Addr#`

- Raw pointers don't work as keys
  - Instead generate a unique value every time host-side array is constructed.

```
data Array sh e = Array … Int … MutVar# … Addr#

getUniqueId :: Array sh e -> IO Int
```

Caching



Other stuff

- Fragmentation
  - Compaction?
- GPU is shared with other process?
  - Serious problems.