# Streaming in Accelerate

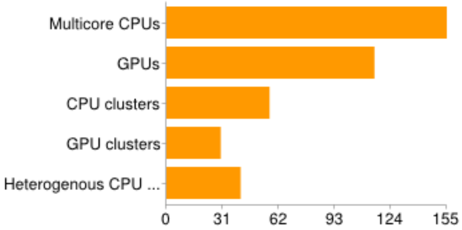
Frederik Meisner Madsen

DIKU
fmma@diku.dk

August 28, 2014

# A short motivation

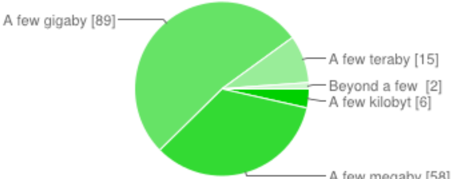
A recent survey on the future of array-oriented computing in Haskell by Manuel Chakravarty[1] - Interesting highlights: **Which type of high-performance hardware are you interested in using?**




[1] http://justtesting.org/post/70881852870/the-future-of-array-oriented-computing-in-haskell-the

# A short motivation

**What is the typical size of your data sets?**



# A short motivation

- Conflict:
  - GPUs excel at computations on lots of data.
    - Not good for small problems due to low occupancy and overhead of GPU initialization.
  - Memory capacity is limited.
    - GeForce GTX 770: 4 GB device memory.
- Problem: Run out of memory fast.
- Solution: Manifest only when absolutely necessary.

## Streaming

- Language design choice:
  - Manual streaming:
    - Easy for language implementer (do nothing).
    - Nightmare for programmer (manage buffers, scheduling, tied to specific platform).
  - Language-integrated streaming:
    - Nightmare for language implementer (to get right).
    - Easy for programmer.

# NESL

- Forefather to Data Parallel Haskell.
- Based on (SIMD) vector-model, suitable for GPUs.
- Small language with formal (time) cost model, suitable for research.
  - Work and step.
- Most innovative feature: Vectorization of nested data parallelism.
  - Theoretical ideal asymptotic complexity.
    - With the right tricks: Scattered segment descriptors.
  - Space in order of <u>exposed</u> parallelism.
    - Naive matrix mult. requires $O(N^3)$ space.
    - Even moderate sized data sets can run into space problems.
    - Programmer should not be punished for exposing too much parallelism.

# NESL + streams

- Goal: Language-integrated streaming.
- Two container types: Vectors $[\tau]$ and sequences $\{\sigma\}$.
- Sequences:
  - Semantically identical to vectors.
  - Processed one element at a time from first to last element.
    - No rewinding (time).
    - Reuse same memory for each element (space).
  - Backend is free to choose a chunk size.
    - Data parallelism.
    - CPU: Chunk size = 1.
    - GPU: Chunk size = 100.000.
- Sequence of vectors $\{[\tau]\}$:
  - Many important applications: Lines of a file, frames of a video.
  - Chunk must be able to grow dynamically to contain at least one vector.

# NESL syntax (simplified)

$$\boxed{op :: \tau \to \tau}$$

$$
\begin{array}{lcl}
e & ::= & constant \\
 & | & x \\
 & | & \textbf{let } x = e_1 \textbf{ in } e_2 \\
 & | & (e_1, ..., e_k) \mid e.k \\
 & | & op\ e \\
 & | & [e_1 : x \textbf{ in } e_2] \\
 & & \\
 & & (= map\ (\lambda x.e_1)\ e_2)
\end{array}
$$

$$
\begin{array}{lcl}
+ & :: & (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Int} \\
 & \vdots & \\
mkvec^k_\tau & :: & \overbrace{(\tau, .., \tau)}^{k} \to [\tau] \\
length_\tau & :: & [\tau] \to \mathsf{Int} \\
!_\tau & :: & ([\tau], \mathsf{Int}) \to \tau \\
concat_\tau & :: & [[\tau]] \to [\tau] \\
partition_\tau & :: & ([\tau], [\mathsf{Int}]) \to [[\tau]] \\
scan & :: & [\mathsf{Int}] \to [\mathsf{Int}] \\
sum & :: & [\mathsf{Int}] \to \mathsf{Int} \\
 & \vdots &
\end{array}
$$

# NESL + streams syntax

$$\boxed{sop :: \sigma \to \sigma}$$

$$\begin{array}{lll} e & ::= & \cdots \\ & \mid & sop\ e \\ & \mid & \{e_1 : x \ \mathbf{in}\ e_2\} \end{array}$$

$$\begin{array}{lll} mkseq^k_\sigma & :: & \overbrace{(\sigma, .., \sigma)}^{k} \to \{\sigma\} \\ sconcat_\sigma & :: & \{\{\sigma\}\} \to \{\sigma\} \\ flagpart_\sigma & :: & (\{\sigma\}, \{\mathsf{Bool}\}) \to \{\{\sigma\}\} \\ sscan & :: & \{\mathsf{Int}\} \to \{\mathsf{Int}\} \\ ssum & :: & \{\mathsf{Int}\} \to \mathsf{Int} \\ tab & :: & \{\tau\} \to [\tau] \\ seq & :: & [\tau] \to \{\tau\} \\ & \vdots & \end{array}$$

# NESL + streams

- Sequence operations can simulate almost all vector operations, except random-access and constant-time length:

$$concat_{\mathsf{Int}}(e)$$

$$\equiv$$

$$tab_{\mathsf{Int}}(sconcat_{\mathsf{Int}}(\{seq_{\mathsf{Int}}(x) : x \textbf{ in } seq_{[\mathsf{Int}]}(e)\}))$$

  - Same time complexity in cost model.
- Similarly, sequence comprehension can simulate vector comprehension.
- After eliminating redundant syntax, the only vector-related syntactic constructions are:

$$(!_\tau),\ length_\tau,\ tab_\tau \text{ and } seq_\tau.$$

# NESL + streams

- Variables in sequence comprehension body cannot have sequence type:
  - Otherwise, sequence is reused once per element.
  - Easily checked by type system.
  - Workarounds:
    - Explicit tabulation using $tab_\tau$.
    - Explicit recomputation by inlining.
  - Instead of silently making the choice in the compiler, resulting in two significantly different time/space complexities, the programmer is forced to make the choice.
- Vectorization:
  - Note: No vector-comprehension.
  - No scattered segment descriptors for sequences.
    - Type system forbids the cases where it would be needed.

# NESL + streams

- Memory cost model.
  - Work and step analogue.
- Programming experience is almost as NESL.
  - Forced to make decisions about what should be vectors and what should be sequences.
    - ... but without a particular backend in mind.
  - Easy to reason about space.
- Open issues:
  - Feasible in practice?
    - Large constants in time complexity.
  - Schedulability / rate analysis.
    - Type system does not reject

      $$\{sum(xs)\} +\!\!+\, xs.$$

    - Note that

      $$xs +\!\!+ \{sum(xs)\}$$

      is perfectly fine.
  - Cost preservation theorem.

# Accelerate

- DSL embedded in Haskell.
- Based on multi-dimensional array operations.
- Flat data parallelism (for now).
  - Regular nesting, the rows of a matrix all have the same length.
- GPU backend.

# Accelerate

$$\begin{array}{rcl} \text{Scalar } a & = & \text{Array Z } a \\ \text{Vector } a & = & \text{Array (Z :. Int) } a \\ \text{Matrix } a & = & \text{Array (Z :. Int :. Int) } a \end{array}$$

$$\begin{array}{lcl} \mathit{generate} & :: & sh \to (sh \to a) \to \text{Array } sh\ a \\ \mathit{map} & :: & (a \to b) \to \text{Array } sh\ a \to \text{Array } sh\ b \\ \mathit{zipWith} & :: & (a \to b \to c) \to \text{Array } sh\ a \to \text{Array } sh\ b \to \text{Array } sh\ c \\ \mathit{scanl} & :: & (a \to a \to a) \to a \to \text{Vector } a \to \text{Vector } a \\ \mathit{fold} & :: & (a \to a \to a) \to a \to \text{Array } (sh :. \text{Int})\ a \to \text{Array } sh\ a \end{array}$$

- Omitted: Everything wrapped in Acc.
  - Operations construct AST terms, $\mathit{run} :: \text{Acc } a \to a$.
  - All sharing is lost initially, recovered using De Bruijn indices.

# Accelerate + streams

- What I hope to gain:
  - Data parallel streaming feasible in practice?
  - Contribution to Accelerate.
- Main challenge:
  - NESL + streams: Experimental toy language, designed from scratch with streaming in mind.
  - Accelerate + streams: Add streaming to real-world language.
- A multi-dimensional array in Accelerate is almost the same as a flat vector in NESL.
  - Both are fully manifest.
  - Data has identical representations in memory.
- Accelerate is similar to NESL.
  - Operations take array extents as additional arguments.
  - Specialized array operations (e.g. stencil).
  - Does not have nested vectors or vector-comprehensions.

# Accelerate + streams - Take 1

- Like NESL + streams, introduce sequence container surface type.
- Unlike NESL + streams, do not eliminate existing vector operations.
  - Accelerate is already optimized for high performance.
  - Existing benchmarks should not become worse.
  - Breaks other backends.
  - Reimplementing everything defeats the purpose of using Accelerate in the first place.
- Sequence = Ordinary Haskell list.
  - $\{\text{Int}\} \simeq [\text{Scalar Int}] = [\text{Array Z Int}]$.
  - $\{[\tau]\} \simeq [\text{Array } (sh :. \tau)\ \tau]$.
  - Get streaming from Haskell's lazy evaluation strategy.

# Accelerate + streams - Take 1

$$
\begin{aligned}
& \textit{type}\ \mathsf{A} = \mathsf{Array} \\
\mathit{mapStream} &:: (\mathsf{A}\ sh\ a \rightarrow \mathsf{A}\ sh'\ b) \\
&\rightarrow [\mathsf{A}\ sh\ a] \rightarrow [\mathsf{A}\ sh'\ b] \\
\mathit{zipWithStream} &:: (\mathsf{A}\ sh\ a \rightarrow \mathsf{A}\ sh'\ b \rightarrow \mathsf{A}\ sh''\ c) \\
&\rightarrow [\mathsf{A}\ sh\ a] \rightarrow [\mathsf{A}\ sh'\ b] \rightarrow [\mathsf{A}\ sh''\ c] \\
\mathit{foldStream} &:: (\mathsf{A}\ sh\ a \rightarrow \mathsf{A}\ sh\ a \rightarrow \mathsf{A}\ sh\ a) \\
&\rightarrow \mathsf{A}\ sh\ a \\
&\rightarrow [\mathsf{A}\ sh\ a] \\
&\rightarrow \mathsf{A}\ sh\ a \\
\mathit{toStream} &:: \mathsf{A}\ (sh :. \mathsf{Int})\ a \rightarrow [\mathsf{A}\ sh\ a] \\
\mathit{fromStream} &:: [\mathsf{A}\ sh\ a] \rightarrow (\mathsf{Vector}\ sh, \mathsf{Vector}\ a)
\end{aligned}
$$

# Accelerate CUDA backend

- Execution:

$$\begin{aligned} \mathit{executeOpenAcc} &:: \mathrm{ExecOpenAcc}\ \mathit{aenv}\ \mathit{arrs} \\ &\to \mathit{aenv} \\ &\to \mathrm{CIO}\ \mathit{arrs} \end{aligned}$$

- ExecOpenAcc *aenv arrs*: Executable AST.
  - Sharing recovery.
  - Fusion.
  - CUDA Code generation.
  - *aenv*:
    - Type-level list.
    - Sharing context from previous let bindings.
  - *arrs*: Result type (e.g. Vector Int).

# Accelerate CUDA backend

- CIO = ReaderT Context (StateT State IO)
  - Context: Device properties and execution context.
  - State:
    - Host/device memory associations.
    - Compiled kernel object code.

# Accelerate + streams - Take 1

- *executeOpenAcc* (MapStream *f acc*) *aenv* :: CIO [A *sh a*]:

$$\begin{aligned} &\mathit{executeOpenAcc}\ (\text{MapStream}\ f\ \mathit{acc})\ \mathit{aenv} = \\ &\quad \mathit{do}\quad \mathit{as}\ \leftarrow\ \mathit{executeOpenAcc}\ \mathit{acc}\ \mathit{aenv} \\ &\qquad\qquad \mathit{mapM}\ (\mathit{executeOpenAfun}\ f\ \mathit{aenv})\ \mathit{as} \end{aligned}$$

- Problem: Both state and IO monad are strict, all elements will manifest.
  - Lazy IO?
- Coming up with a solution - 4 failed approaches.

# Accelerate + streams - Swapping types

- Use [CIO *arrs*] instead of CIO [*arrs*]?
- Define new function:

$$\begin{aligned} \mathit{streamOpenAcc} &:: \text{ExecOpenAcc}\ \mathit{aenv}\ [\mathit{arrs}] \\ &\to \mathit{aenv} \\ &\to [\text{CIO}\ \mathit{arrs}] \end{aligned}$$

$$\begin{aligned} &\mathit{streamOpenAcc}\ (\text{MapStream}\ f\ \mathit{acc})\ \mathit{aenv} = \\ &\quad \mathbf{let}\quad s = \mathit{streamOpenAcc}\ \mathit{acc}\ \mathit{aenv} \\ &\quad \mathbf{in}\quad \mathit{map}\ ({\gg}{=}\ \mathit{executeOpenAfun}\ f\ \mathit{aenv})\ s \end{aligned}$$

- For sub-expression of stream type in *executeOpenAcc*, call *streamOpenAcc* instead of recursion.
- Problem: Sharing streams:
  - Add [CIO *arrs*] to sharing context - Each use recomputes.
  - Run stream and add [*arrs*] to sharing context - Tabulation.

# Accelerate + streams - Look ahead

- Run streams immediately:
  - When a stream producer (*toStream*) is encountered:
    - Traverse AST to find it's consumers.
    - Feed all elements immediately.
    - In *fromStream* and *foldStream* nodes, store result somewhere.
  - When a *fromStream* or *foldStream* is encountered, simply fetch the stored result.
- Problems:
  - A seemingly irrelevant AST node may introduce a new bindings required by a later consumer.
  - *zipWithStream* requires two producers feeding elements in lock-step.
    - Traverse AST to find all producers in the same "loop" before starting to feed elements.
  - Many traversals, static analysis.

# Accelerate + streams - Pipes

- Pipes is a popular library designed as a safe replacement to lazy IO.
- Combines effects, streaming and compositionality.
- Use
  $$\text{Pipe } \mathit{aenv}\ \mathit{arrs}\ \text{CIO}\ ()$$
  instead of
  $$\mathit{aenv} \rightarrow \text{CIO}\ [\mathit{arrs}]?$$
- Define new function:
  $$\begin{aligned} \mathit{pipeOpenAcc} \quad &:: \quad \text{ExecOpenAcc } \mathit{aenv}\ [\mathit{arrs}] \\ &\rightarrow \quad \text{Pipe } \mathit{aenv}\ \mathit{arrs}\ \text{CIO}\ () \end{aligned}$$
- Bind *arrs* instead of [*arrs*] in *aenv*. Feed *aenv* multiple times.

# Accelerate + streams - Pipes

- Mapping closed functions:

$$\begin{aligned}&\mathit{pipeOpenAcc}\ (\text{MapStream}\ f\ \mathit{acc}) =\\ &\quad \mathit{pipeOpenAcc}\ \mathit{acc} \rightarrowtail \mathit{mapPipe}\ (\mathit{execClosedAfun}\ f)\end{aligned}$$

- Mapping open functions / let bindings:

$$\begin{aligned}&\mathit{pipeOpenAcc}\ (\text{Alet}\ \mathit{bnd}\ \mathit{bdy}) =\\ &\quad (\mathit{pipeOpenAcc}\ \mathit{bnd} \times \mathit{idPipe}) \rightarrowtail \mathit{mapPipe}\ \mathit{pipeOpenAcc}\ \mathit{bdy}\end{aligned}$$

- $(\times)$ does not exist for Pipe.
  - A pipe may consume and produce in any order.

# Accelerate + streams - Pipes

- Workarounds:
  - Produce (Either $a$ $b$) instead of $(a, b)$.
    - Context becomes a sum type:

$$\text{Pipe } (\textstyle\sum aenv) \; arrs \; \text{CIO } ()$$

    - Buffering.
    - Upstream communication: Reject, request.
  - Define a less general version of Pipe where a value is always produced directly after a value is consumed.
    - Better.

# Accelerate + streams - Pipes

- Problem number 2:

$$
\begin{aligned}
&\mathit{bnd} :: \text{ExecOpenAcc}\ \mathit{aenv}\ [a] \\
&\mathit{bdy} :: \text{ExecOpenAcc}\ \mathit{aenv}\ b \quad (b \neq [\_]) \\
&\mathit{executeOpenAcc}\ (\text{Alet}\ \mathit{bnd}\ \mathit{bdy}) = ??
\end{aligned}
$$

  - Bind the pipe → recomputation.
  - Tabulate pipe, bind the result → tabulation.
  - Make everything a pipe.

# Accelerate + streams - Concurrency

- Scheduling shared streams seems to be the core of the problem.
- Task parallelism is natural for stream transformers.
- Use (MVar *arrs*) instead of [*arrs*]?
- Execute stream operation:
  - Fork stream transformer. Take, compute, put loop.
- Problems:
  - Barrier synchronization for multiple consumers.
  - Synchronize State.
  - Synchronize communication with GPU.
  - ... and not just for streams. A binary array operation must wait if one operand is the result of a stream reduction.

# Accelerate + streams - Take 1 conclusion

- Sharing streams is fundamentally different than sharing arrays.
  - Imposes restrictions on evaluation order.
- Streams as first-class "changes" all other types.
  - E.g. Int could be an integer available now, or an integer available in the future.
  - Not a problem in NESL + stream, compiler transforms everything to streams.
- Stream operations promise too much:
  - Partiality in the language.
  - Silent stream tabulation.

# Accelerate + streams - Take 2

- Separate stream sharing contexts from array sharing context.
- New AST: Loop *lenv arrs*
  - Put all stream operations in Loop.
  - *lenv*: Loop sharing context (type-level list).
  - *arrs*: Result(s) of running the loop.
  - Allow closed loops in Accelerate AST:

$$loop :: \text{Loop } () \; arrs \rightarrow \; \text{Acc } arrs$$

- Sequence = De Bruijn index into *lenv*.
  - $\{\text{Int}\} \simeq \text{Idx } lenv \text{ Int}$.

$$\begin{array}{l} data \; \text{Idx } env \;\; a \;\; where \\ \quad \text{ZeroIdx} :: \text{Idx } (env, a) \; a \\ \quad \text{SuccIdx} :: \text{Idx } env \;\; a \rightarrow \text{Idx } (env, b) \;\; a \end{array}$$

# Accelerate + streams - Take 2

$$
\begin{array}{lcl}
\mathit{emptyLoop} & :: & \mathsf{Loop}\ \mathit{lenv}\ () \\
\mathit{mapStream} & :: & (\mathsf{A}\ \mathit{sh}\ a \rightarrow \mathsf{A}\ \mathit{sh}'\ b) \\
 & \rightarrow & \mathsf{Idx}\ \mathit{lenv}\ (\mathsf{A}\ \mathit{sh}\ a) \\
 & \rightarrow & \mathsf{Loop}\ (\mathit{lenv}, \mathsf{A}\ \mathit{sh}'\ b)\ \mathit{arrs} \\
 & \rightarrow & \mathsf{Loop}\ \mathit{lenv}\ \mathit{arrs} \\
\mathit{zipWithStream} & :: & (\mathsf{A}\ \mathit{sh}\ a \rightarrow \mathsf{A}\ \mathit{sh}'\ b \rightarrow \mathsf{A}\ \mathit{sh}''\ c) \\
 & \rightarrow & \mathsf{Idx}\ \mathit{lenv}\ (\mathsf{A}\ \mathit{sh}\ a) \\
 & \rightarrow & \mathsf{Idx}\ \mathit{lenv}\ (\mathsf{A}\ \mathit{sh}'\ b) \\
 & \rightarrow & \mathsf{Loop}\ (\mathit{lenv}, \mathsf{A}\ \mathit{sh}''\ c)\ \mathit{arrs} \\
 & \rightarrow & \mathsf{Loop}\ \mathit{lenv}\ \mathit{arrs}
\end{array}
$$

# Accelerate + streams - Take 2

$$
\begin{aligned}
\mathit{foldStream} \quad &:: \quad (\mathsf{A}\ \mathit{sh}\ a \rightarrow \mathsf{A}\ \mathit{sh}\ a \rightarrow \mathsf{A}\ \mathit{sh}\ a) \\
&\rightarrow \quad \mathsf{A}\ \mathit{sh}\ a \\
&\rightarrow \quad \mathsf{Idx}\ \mathit{lenv}\ (\mathsf{A}\ \mathit{sh}\ a) \\
&\rightarrow \quad \mathsf{Loop}\ \mathit{lenv}\ \mathit{arrs} \\
&\rightarrow \quad \mathsf{Loop}\ \mathit{lenv}\ (\mathit{arrs}, \mathsf{A}\ \mathit{sh}\ a) \\
\mathit{toStream} \quad &:: \quad \mathsf{A}\ (\mathit{sh} :.\ \mathsf{Int})\ a \\
&\rightarrow \quad \mathsf{Loop}\ (\mathit{lenv}, \mathsf{A}\ \mathit{sh}\ a)\ \mathit{arrs} \\
&\rightarrow \quad \mathsf{Loop}\ \mathit{lenv}\ \mathit{arrs} \\
\mathit{fromStream} \quad &:: \quad \mathsf{Idx}\ \mathit{sh}\ a \\
&\rightarrow \quad \mathsf{Loop}\ \mathit{lenv}\ \mathit{arrs} \\
&\rightarrow \quad \mathsf{Loop}\ \mathit{lenv}\ (\mathit{arrs}, (\mathsf{Vector}\ \mathit{sh}, \mathsf{Vector}\ a))
\end{aligned}
$$

# Accelerate + streams - Take 2

- Loop terminates when the first producer is exhausted (like *zip* in Haskell or vector-comprehension in NESL).
- Status: Reference interpreter works, confident that CUDA backend will work too.
- Difference from take 1:
  - Same operations.
  - Cannot let bind new arrays in Loop.
  - Move array let-bindings to just before Loop:
    - If not possible, array is a reduction of one of the streams of the loop.
    - Move to after Loop.
    - If not possible, a stream of the loop depends on a reduction of the loop $\rightarrow$ exactly when partiality or silent stream tabulation was required in take 1.
- This is as far as I got.

# Accelerate + streams - Take 2

$$\sum_{i=1}^{n} \log i$$

```
iota :: Int -> Acc (Vector Int)
iota n = generate (index1 (constant n)) unindex1

-- Take 1
logsum :: Int -> Acc (Scalar Float)
logsum n = foldStream (zipWith (+)) (use (fromList Z [0.0]))
         $ mapStream (map (log . fromIntegral . (+1)))
         $ toStream (iota n)

-- Take 2
logsum :: Int -> Acc (Scalar Float)
logsum n = asnd $ loop
         $ toStream (iota n)
         $ mapStream (map (log . fromIntegral . (+1))) ZeroIdx
         $ foldStream (zipWith (+)) (use (fromList Z [0.0])) ZeroIdx
         $ emptyLoop
```

# Accelerate + streams - Take 2 conclusion

- Essentially a sub-language for sequences.
- Many open questions:
  - Surface language with sharing recovery.
    - Should be easy enough.
  - More stream operations.
    - Scan, filter.
  - Generalize to chunk size $> 1$.
    - Treat scalars (fixed-buffer) different from non-scalars (dynamic buffer) in loop context.
    - Variable number of elements produced, loops become dataflow networks.
  - Nested data parallelism.
    - Lifted loops.
    - Nested loops.

# Conclusion

- For GPUs, streaming is a necessity.
  - Language-integrated streaming is preferable in high-level languages.
- Language-integrated first-class streaming is all or nothing.
  - Nothing is a stream or everything is a stream.
  - NESL: Everything is lifted to sequence space.
  - Accelerate: Streams are confined to a sub-language.
    - Closed non-suspendable loop. No communication across loops.
- Data parallel streaming feasible in practice?
  - Perhaps, but difficult to get right.
  - No benchmarks yet.