Haskell Beats C using Generalized Stream Fusion

Christoph-Simon Senjak, Christian Neukirchen

Lehr- und Forschungseinheit für Theoretische Informatik
Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstr.67, 80538 München

PUMA & RiSE Workshop 2014
Exploiting Vector Instructions with Generalized Stream Fusion

Geoffrey Mainland  
Microsoft Research Ltd  
Cambridge, England  
gmainlan@microsoft.com

Roman Leshchinskiy  
rl@cse.unsw.edu.au

Simon Peyton Jones  
Microsoft Research Ltd  
Cambridge, England  
simonpj@microsoft.com

(ICFP 2013)
Reviewing stream fusion

- Want to write *efficient* code, yet still use map, filter, zip:

  ```haskell
dotp :: List Double \rightarrow\n List Double \rightarrow\n Double
  dotp v w = foldl (+) 0 (zipWith (*) v w)
  ```

- Eliminate immediate data structures

- Fusion is easier on non-recursive co-structures: stream fusion (Coutts *et al.* 2007, Coutts 2010)

```haskell
data Stream a where
  Stream :: (s \rightarrow\n Step s a) \rightarrow\n s \rightarrow\n Stream a

data Step s a = Yield a s
  | Skip s
  | Done
```
Converting between lists and streams

\[
\begin{align*}
\text{stream} &:: [a] \rightarrow \text{Stream } a \\
\text{stream } xs & = \text{Stream } \text{uncons } xs \\
\text{where} \text{uncons } [ ] & = \text{Done} \\
\text{uncons } (x : xs) & = \text{Yield } x \times xs \\
\end{align*}
\]

\[
\begin{align*}
\text{unstream} &:: \text{Stream } a \rightarrow [a] \\
\text{unstream } (\text{Stream } \text{next } s) & = \text{unfold } s \\
\text{where} \text{unfold } s & = \text{case } \text{next } s \text{ of} \\
\text{Done} & \rightarrow [ ] \\
\text{Skip } s' & \rightarrow \text{unfold } s' \\
\text{Yield } x \times s' & \rightarrow x : \text{unfold } s' \\
\end{align*}
\]
Operations on streams

- Writing map for streams:

  \[ \text{mapS} :: (a \to b) \to \text{Stream} a \to \text{Stream} b \]
  \[ \text{mapS} \ f \ (\text{Stream} \ \text{step} \ s) = \text{Stream} \ \text{step'} \ s \]
  \[ \text{where} \ \text{step'} \ s = \text{case} \ \text{step} \ s \ of \]
  \[ \quad \text{Yield} \ x \ s' \to \text{Yield} \ (f \ x) \ s' \]
  \[ \quad \text{Skip} \ s' \to \text{Skip} \ s' \]
  \[ \quad \text{Done} \to \text{Done} \]

- Writing map for lists:

  \[ \text{map} :: (a \to b) \to \text{List} a \to \text{List} b \]
  \[ \text{map} \ f = \text{unstream} \circ \text{mapS} \ f \circ \text{stream} \]

- We can define foldl, filter and zipWith using streams similarly.
GHC can optimize code using rewrite rules such as “stream ◦ unstream = id”.

\[
\text{map } f \circ \text{map } g \\
\equiv \text{unstream} \circ \text{mapS} \ f \circ \text{stream} \circ \text{unstream} \circ \text{mapS} \ g \circ \text{stream} \\
\{ \text{by rewriting} \} \\
\equiv \text{unstream} \circ \text{mapS} \ f \circ \text{mapS} \ g \circ \text{stream} \\
\{ \text{by inlining} \} \\
\equiv \text{unstream} \circ \text{mapS} \ (f \circ g) \circ \text{stream}
\]
For SIMD, we want to operate on multiple values in parallel (e.g. for SSE, two doubles or four floats)

Type class to abstract SIMD:

```haskell
data Multi a
multiplicity :: Multi a → Int
multireplicate :: a → Multi a
multimap :: (a → a) → Multi a → Multi a
multifold :: (b → a → b) → b → Multi a → b
multizipWith :: (a → a → a) → Multi a → Multi a → Multi a
```
**Streaming Multis**

- Make streams that pass \( n \) values at once: Multis
- Can either let the producer or consumer dictate which representation is used

```hs
data Either a b = Left a | Right b

type MultisP a = Stream (Either a (Multi a))

data MultisC a where
    MultisC :: (s → Step s (Multi a))
        → (s → Step s a)
        → s
        → MultisC a

type Multis a = Either (MultisC a) (MultisP a)
```

- MultisC is preferred, but not always possible to use
Bundling streams

- Different functions prefer different data representations
- Bundle all of them into a single data type:

```haskell
data Bundle a =
    Bundle { sElems :: Stream a, sMultis :: Multis a, ... }
```

- Functions should use only one representation; only this one needs to be computed. Compiler picks appropriate one (first pattern match) → **Generalized stream fusion**
- Generalized stream fusion is implemented as Haskell library, which GHC optimizes away completely (if used correctly); it could also be a compiler immediate language.
Implementation

- Add SSE support to GHC
- Implement Multi type that uses SSE for primitives
- Modify vector library to use generalized stream fusion and bundles
- Modify DPH library to use new vector library/bundles
4 benchmarks are given

- Single-thread performance of double-precision dot product
- Percentage speedup of existing Haskell libraries
- Performance of Haskell vs C vs C++: Gaussian radial basis function
- Performance of double-precision dot product, multithreaded
Naïve implementation:

```c
double cddotp(double* u, double* v, int n)
{
    double s = 0.0;
    int i;
    for (i = 0; i < n; ++i)
        s += u[i] * v[i];
    return s;
}
```

Using SSE but not pre-fetching:

```c
#include <xmmmintrin.h>
#define VECTOR_SIZE 2
typedef double v2sd __attribute__((vector_size(sizeof(double)*VECTOR_SIZE)));
union d2v
{
    v2sd v;
    double d[VECTOR_SIZE];
};

double ddotp(double* u, double* v, int n)
{
    union d2v d2s = {0.0, 0.0};
    double s;
    int i;
    int m = n & (~VECTOR_SIZE);
    for (i = 0; i < m; i += VECTOR_SIZE)
        d2s.v += (*((v2sd*) (u+i)))*((v2sd*) (v+i));
    s = d2s.d[0] + d2s.d[1];
    for (; i < n; ++i)
        s += u[i] * v[i];
    return s;
}
```
Hand-written C implementations almost equal \( \rightarrow \) GCC’s optimization.

Haskell outperforms GCC’s vectorizer.

After L3-cache is exhausted, Haskell can compete with GotoBLAS.
Single-thread performance of double-precision dot product

However, not tested with Intel’s C compiler, which probably optimizes better.

Haskell uses prefetching instructions, which are not used in the C example.
5.1 Prefetching and loop unrolling

Why is Haskell so fast? Because we have exploited the high-level stream-fusion framework to embody two additional optimizations: prefetching and loop unrolling.

To see how easy it is to integrate SIMD instruction into existing programs, we rewrote a number of functions from various packages and implemented the expectation-maximization (EM) algorithms. The speedup of the rewritten versions is shown in Figure 8.

Not only can the client of our modified library write vector functions that contains multiple values of data, we know exactly what memory access pattern will be used—each element of the vector will be accessed in linear order. However, using our modified library, these prefetch hints are implemented using prefetch memory that the program plans to use in the future. In doing so, we give the CPU a hint about memory access patterns, telling it to prefetch instructions as it yields each item to the stream, this knowledge by executing prefetch instructions as it yields each item to the stream-fusion framework to embody two additional optimizations: prefetching and loop unrolling.

The function that performs this conversion, stream, is adapted from the StatisticalMethods benchmark is adapted from the examples included in the statistics package and implements the expectation-maximization (EM) algorithms. The speedup of the rewritten versions is shown in Figure 8.

The first six benchmarks consist almost entirely of numeric operations for which SSE version are available; correspondingly, we could only run it on very small data sets. Nevertheless, we were able to gain at least a minimum of a few percentage points even on these difficult-to-vectorize benchmarks just by picking some low-hanging fruit. Notably, we were unable to improve the run time of mixture and quickhull need non-numeric data options and data-dependent control flow → hard to vectorize.

Still an improvement in any case.

- mixture
- quickhull

Speedup of benchmarks from using modified vector

![Graph showing speedup of benchmarks](image)

- mixture and quickhull need non-numeric data options and data-dependent control flow → hard to vectorize.
- Still an improvement in any case.
Performance of Haskell and C Gaussian RBF implementations

- $K(\vec{x}, \vec{y}) = \exp(-\nu \|\vec{x} - \vec{y}\|^2)$
- With BLAS either multiple passes, or intermediate structures.
- C cannot perform fusion of array operations. C++ can.
Performance of Haskell and C++ Gaussian RBF implementations

- C++ uses const references and expression templates (≈ inlined code).
- In Haskell, you do not have to care about it.
- Room for improvement of the Haskell library.
Performance of double-precision dot product implementations

- DPH can automagically utilize multiple cores.
- Still, there appears no relevant speedup with $> 4$ threads.
Summary

- Generalized Stream Fusion is a great way to optimize numerical Haskell code
- Numerical Haskell code does not need to look ugly
- GHC code for numerical programs can compete with GCC
- GHC is very flexible and has a very generic optimizer
- Haskell’s abstraction allows to take advantage of these optimizations at all levels (DPH)