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

Published version has a less sensational title . . .

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:

```
dotp :: List Double → List Double → 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)

```
data Stream a where
  Stream :: (s → Step s a) → s → Stream a
data Step s a = Yield a s
              | Skip s
              | Done
```

Converting between lists and streams

```
stream :: [a] → Stream a
stream xs = Stream uncons xs
   where uncons [ ] = Done
         uncons (x : xs) = Yield x xs

unstream :: Stream a → [a]
unstream (Stream next s) = unfold s
   where unfold s = case next s of
                      Done → [ ]
                      Skip s' → unfold s'
                      Yield x s' → x : unfold s'
```

Operations on streams

- Writing map for streams:

```
mapS :: (a → b) → Stream a → Stream b
mapS f (Stream step s) = Stream step' s
  where step' s = case step s of
                    Yield x s' → Yield (f x) s'
                    Skip s' → Skip s'
                    Done → Done
```

- Writing map for lists:

```
map :: (a → b) → List a → List b
map f = unstream ∘ mapS f ∘ stream
```

- We can define foldl, filter and zipWith using streams similarly.

Map fusion for free

- GHC can optimize code using rewrite rules such as "stream ∘ unstream = id".

$$
\begin{aligned}
&\text{map f} \circ \text{map g} \\
&= \text{unstream} \circ \text{mapS f} \circ \text{stream} \circ \text{unstream} \circ \text{mapS g} \circ \text{stream} \\
&\quad \{-\ \textit{by rewriting}\ -\} \\
&= \text{unstream} \circ \text{mapS f} \circ \text{mapS g} \circ \text{stream} \\
&\quad \{-\ \textit{by inlining}\ -\} \\
&= \text{unstream} \circ \text{mapS (f} \circ \text{g)} \circ \text{stream}
\end{aligned}
$$

Single Instruction Multiple Data

- 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:

  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

```
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:

  ```
  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

Evaluation

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

Single-thread performance of double-precision dot product

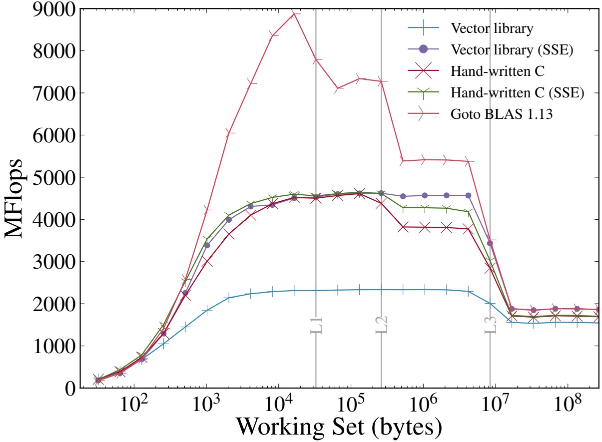
Naïve implementation:

```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 prefetching:

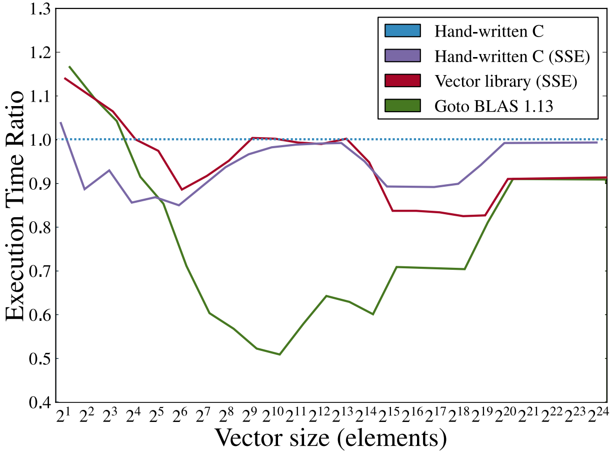
```c
#include <xmmintrin.h>

#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;
}
```

Single-thread performance of double-precision dot product



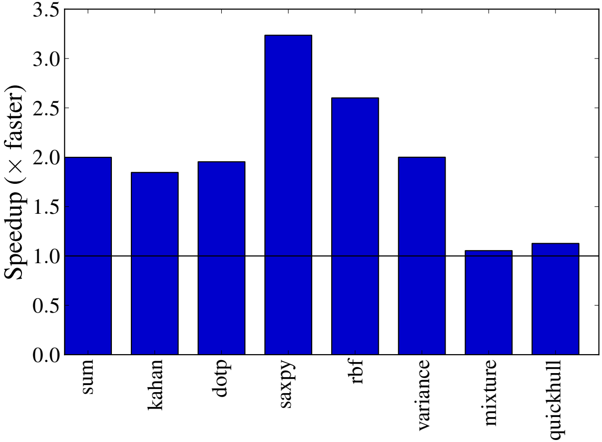
- Hand-written C implementations almost equal → GCC's optimization.
- Haskell outperforms GCC's vectorizer.
- After L3-cache is exhausted, Haskell can compete with GotoBLAS.

Single-thread performance of double-precision dot product



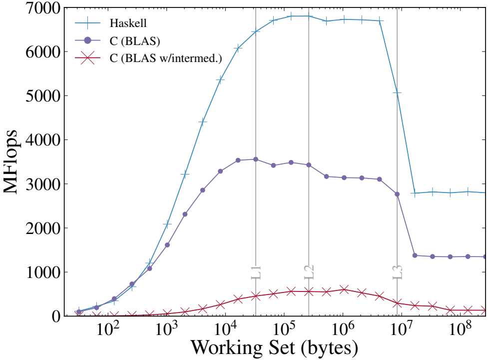
- However, not tested with Intel's C compiler, which probably optimizes better.
- Haskell uses prefetching instructions, which are not used in the C example.

Speedup of benchmarks from using modified vector



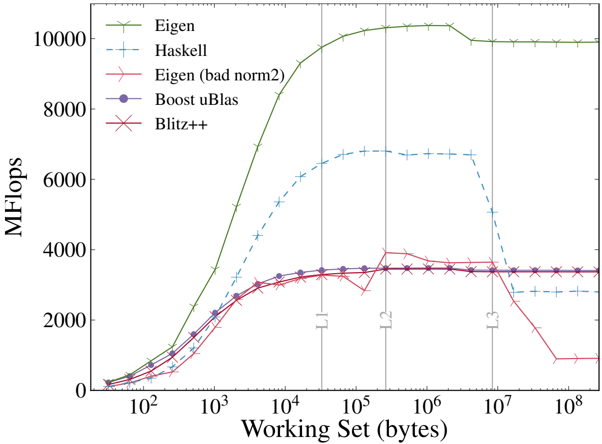
- `mixture` and `quickhull` need non-numeric data options and data-dependent control flow → hard to vectorize.
- Still an improvement in any case.

Performance of Haskell and C Gaussian RBF implementations



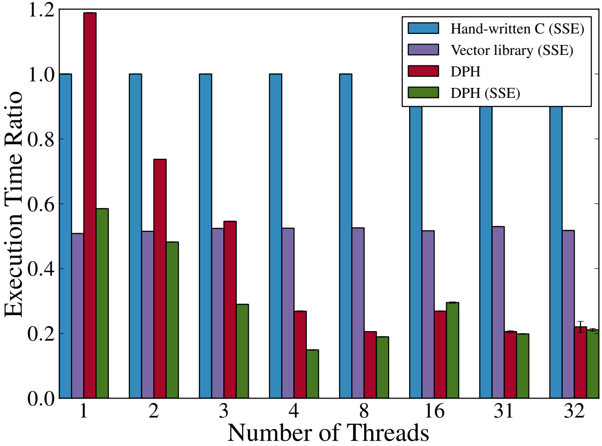
- $K(\vec{x}, \vec{y}) = \exp(-v\|\vec{x} - \vec{y}\|^2)$
- With BLAS either multiple passes, or intermediate structures.
- C cannot perform fusion of array operations. C++ can.

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)