Haskell Beats C Using Generalized Stream Fusion

Geoffrey Mainland

Microsoft Research Ltd

Chalmers, December 2012

Joint work with Roman Leshchinskiy, Simon Peyton Jones, and Simon Marlow.

The challenge...

- Write high-level, *high-performance*, sequence-processing code. Think numerical code and vectors.
- In particular, allow compositional code without sacrificing performance.
- Should be able to take advantage of special-case operations, e.g., bulk memory copies and SIMD instructions.

The plan

- Overview of streams and stream fusion (old news).
- *Generalized* stream fusion. Simultaneously maintain several representations, and pick the best one depending on operations performed.
- Craft a representation tailored for computing with SSE instructions.
- Produce efficient numerical code from Haskell, i.e., how to make Haskell beat C!

Fusion

$$\mathsf{map}\ f \circ \mathsf{map}\ g$$

Fusion

$$\mathsf{map}\ \mathsf{f} \circ \mathsf{map}\ \mathsf{g} \equiv \mathsf{map}\ (\mathsf{f} \circ \mathsf{g})$$

Stream fusion

- Recursive functions are hard for the compiler to optimize.
- So turn recursive functions into non-recursive functions via a change in representation.
- Instead of working with lists or vectors, work with Streams.

Stream fusion

```
data Stream a where
  Stream :: (s → Step s a) → s → Int → Stream a

data Step s a = Yield a s
              | Skip s
              | Done
```

map, now with Streams

$$\mathsf{map} :: (\mathsf{a} \to \mathsf{b}) \to \mathsf{Vector\ a} \to \mathsf{Vector\ b}$$
$$\mathsf{map\ f} = \mathsf{unstream} \circ \mathsf{map_s\ f} \circ \mathsf{stream}$$

map, now with Streams

```
map :: (a → b) → Vector a → Vector b
map f = unstream ∘ map_s f ∘ stream

map_s :: (a → b) → Stream a → Stream b
map_s 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       → z
```

Fusing maps

$$\begin{aligned}&\mathsf{map}\ f \circ \mathsf{map}\ g \equiv \\ &\quad \mathsf{unstream} \circ \mathsf{map}_s\ f \circ \mathsf{stream} \circ \mathsf{unstream} \circ \mathsf{map}_s\ g \circ \mathsf{stream}\end{aligned}$$

Fusing maps

$$\mathsf{map}\ f \circ \mathsf{map}\ g \equiv$$
$$\quad \mathsf{unstream} \circ \mathsf{map}_s\ f \circ \mathsf{stream} \circ \mathsf{unstream} \circ \mathsf{map}_s\ g \circ \mathsf{stream}$$

$$\mathsf{map}\ f \circ \mathsf{map}\ g \equiv$$
$$\quad \mathsf{unstream} \circ \mathsf{map}_s\ f \circ \mathsf{map}_s\ g \circ \mathsf{stream}$$

Fusing maps

$$\mathsf{map}\ \mathsf{f} \circ \mathsf{map}\ \mathsf{g} \equiv$$
$$\quad \mathsf{unstream} \circ \mathsf{map}_\mathsf{s}\ \mathsf{f} \circ \mathsf{stream} \circ \mathsf{unstream} \circ \mathsf{map}_\mathsf{s}\ \mathsf{g} \circ \mathsf{stream}$$

$$\mathsf{map}\ \mathsf{f} \circ \mathsf{map}\ \mathsf{g} \equiv$$
$$\quad \mathsf{unstream} \circ \mathsf{map}_\mathsf{s}\ \mathsf{f} \circ \mathsf{map}_\mathsf{s}\ \mathsf{g} \circ \mathsf{stream}$$

We get $\mathsf{map}\ \mathsf{f} \circ \mathsf{map}\ \mathsf{g} \equiv \mathsf{map}\ (\mathsf{f} \circ \mathsf{g})$ *for free*!

Fusing vector dot product

$$\mathsf{dotp} :: \mathsf{Vector\ Double} \to \mathsf{Vector\ Double} \to \mathsf{Double}$$
$$\mathsf{dotp\ v\ w} = \mathsf{sum}\ (\mathsf{zipWith}\ (*)\ \mathsf{v\ w})$$

Fusing vector dot product

$$\mathsf{dotp} :: \mathsf{Vector\ Double} \to \mathsf{Vector\ Double} \to \mathsf{Double}$$
$$\mathsf{dotp\ v\ w} = \mathsf{sum}\ (\mathsf{zipWith}\ (*)\ \mathsf{v\ w})$$

$$\mathsf{zipWith} :: (\mathsf{a} \to \mathsf{b} \to \mathsf{c}) \to \mathsf{Vector\ a} \to \mathsf{Vector\ b} \to \mathsf{Vector\ c}$$
$$\mathsf{zipWith\ f\ v\ w} = \mathsf{unstream}\ (\mathsf{zipWith}_{\mathsf{stream}}\ \mathsf{f}\ (\mathsf{stream\ v})\ (\mathsf{stream\ w}))$$

$$\mathsf{sum} :: \mathsf{Num\ a} \Rightarrow \mathsf{Vector\ a} \to \mathsf{a}$$
$$\mathsf{sum\ v} = \mathsf{foldl'}_{\mathsf{stream}}\ 0\ (+)\ (\mathsf{stream\ v})$$

Fusing vector dot product

$$
\begin{aligned}
&\mathsf{dotp} :: \mathsf{Vector\ Double} \to \mathsf{Vector\ Double} \to \mathsf{Double} \\
&\mathsf{dotp} \equiv \mathsf{sum}\ (\mathsf{zipWith}\ (*)\ \mathsf{v}\ \mathsf{w}) \\
&\qquad \equiv \mathsf{foldl'}_{\mathsf{stream}}\ 0\ (+)\ (\mathsf{stream}\ (\mathsf{unstream} \\
&\qquad\qquad (\mathsf{zipWith}_{\mathsf{stream}}\ (+)\ (\mathsf{stream}\ \mathsf{v})\ (\mathsf{stream}\ \mathsf{w})))) \\
&\qquad \equiv \mathsf{foldl'}_{\mathsf{stream}}\ 0\ (+) \\
&\qquad\qquad (\mathsf{zipWith}_{\mathsf{stream}}\ (+)\ (\mathsf{stream}\ \mathsf{v})\ (\mathsf{stream}\ \mathsf{w}))
\end{aligned}
$$

Fusing vector dot product: $\mathsf{foldl'}_{\mathsf{stream}}$

```
foldl'_stream :: (a → b → a) → a → Stream b → a
foldl'_stream f z (Stream step s) = loop z s
  where
    loop z s = z `seq`
      case step s of
        Yield x s' → loop (f z x) s'
        Skip s'    → loop z s'
        Done       → z
```

Fusing vector dot product: $\mathrm{zipWith}_{\mathrm{stream}}$

```
zipWithstream :: (a → b → c) → Stream a → Stream b → Stream c
zipWithstream f (Stream stepa sa na) (Stream stepb sb nb) =
    Stream step (sa, sb, Nothing) (min na nb)
  where
    step (sa, sb, Nothing) =
      case stepa sa of
        Yield x sa' → Skip (sa', sb, Just x)
        Skip sa'    → Skip (sa', sb, Nothing)
        Done        → Done

    step (sa, sb, Just x) =
      case stepb sb of
        Yield y sb' → Yield (f x y) (sa, sb', Nothing)
        Skip sb'    → Skip (sa, sb', Just x)
        Done        → Done
```

dotp after inlining etc.

```
dotp (Vector n u) (Vector m v) = loop 0.0 0 0
  where
    loop z i j
      | i < n ∧ j < m = loop (z + u !! i * v !! j) (i + 1) (j + 1)
      | otherwise    = z
```

dotp inner loop

```
.LBB2_3:
    movsd       (%rcx), %xmm0
    mulsd       (%rdx), %xmm0
    addsd       %xmm0, %xmm1
    addq        $8, %rcx
    addq        $8, %rdx
    decq        %rax
    jne .LBB2_3
```

Stream fusion limitations

```
data Stream a where
  Stream :: (s → Step s a) → s → Int → Stream a

data Step s a = Yield a s
              | Skip s
              | Done
```

- How to support SSE instructions in this framework?
- Efficient implementations of append, replicate?

Generalizing Stream Fusion

- No single “best” stream representation.

Generalizing Stream Fusion

- No single "best" stream representation.
- *Bundle* multiple stream representations together.

```
data Bundle a = Bundle
  { sSize    :: Size
  , sElems   :: Stream a
  , sChunks  :: Stream (Chunk a)
  , sMultis  :: Multis a
  }
```

Generalizing Stream Fusion

- No single "best" stream representation.
- *Bundle* multiple stream representations together.
- Stream consumer chooses most advantageous representation.

```
data Bundle a = Bundle
  { sSize    :: Size
  , sElems   :: Stream a
  , sChunks  :: Stream (Chunk a)
  , sMultis  :: Multis a
  }
```

Generalizing Stream Fusion

- No single "best" stream representation.
- *Bundle* multiple stream representations together.
- Stream consumer chooses most advantageous representation.
- Identical semantics, different cost models.

```
data Bundle a = Bundle
  { sSize   :: Size
  , sElems  :: Stream a
  , sChunks :: Stream (Chunk a)
  , sMultis :: Multis a
  }
```

Using SSE instructions in GHC

- Add support for SSE primitives to code generator(s).
- Add boxed SSE types: associated type Multi a.
- Fix the register allocator.
- High-level interface—via generalized streams and the `vector` library.

Stream representation for SSE computation

```
data Either a b = Left a | Right b
type Multis a = Stream (Either a (Multi a))
```

Stream representation for SSE computation

data Either a b = Left a | Right b

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

- Stream consumer has to take what it can get.

Stream representation for SSE computation

```
data Either a b = Left a | Right b
type Multis a = Stream (Either a (Multi a))
```

- Stream consumer has to take what it can get.
- Fine for map or fold...

Stream representation for SSE computation

```
data Either a b = Left a | Right b
type Multis a = Stream (Either a (Multi a))

msum_s :: (Num a, Num (Multi a)) => Multis a -> a
msum_s (Stream step s _) = loop 0.0 0.0 s
  where
    loop summ sum1 s =
      case step s of
        Yield (Left x)  s' -> loop summ       (sum1 + x) s'
        Yield (Right y) s' -> loop (summ + y) sum1       s'
        Skip            s' -> loop summ       sum1       s'
        Done               -> multifold (+) sum1 summ
```

Stream representation for SSE computation

data Either a b = Left a | Right b

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

What about zipWith?

Stream representation for SSE computation, mark II

```
data Multis a where
  Multis :: (s → Step s (Multi a))
         → (s → Step s a)
         → s
         → Multis a
```

- Stream consumer gets to choose what it gets.
- Works for zipWith, map, fold...
- What happens when appending two streams?

Stream representation for SSE computation

```
data MultiStream a where
  MultiStream :: (s → Step s (Multi a))
              → (s → Step s a)
              → s
              → MultiStream a
```

- But stream operations may "mix" scalars and Multi's, e.g., append.

Stream representation for SSE computation

```
data MultiStream a where
  MultiStream :: (s → Step s (Multi a))
              → (s → Step s a)
              → s
              → MultiStream a
```

- But stream operations may "mix" scalars and Multi's, e.g., append.
- Some operations, like fold, are agnostic to "mixing." Others, like zipWith, are not.

Stream representation for SSE computation

```
data MultiStream a where
  MultiStream :: (s → Step s (Multi a))
              → (s → Step s a)
              → s
              → MultiStream a
```

- But stream operations may "mix" scalars and Multi's, e.g., append.
- Some operations, like fold, are agnostic to "mixing." Others, like zipWith, are not.

```
type Multis m a = Either (MultiStream m a)
                         (Stream m (Either a (Multi a)))
```

A more difficult challenge

```
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;
}
```

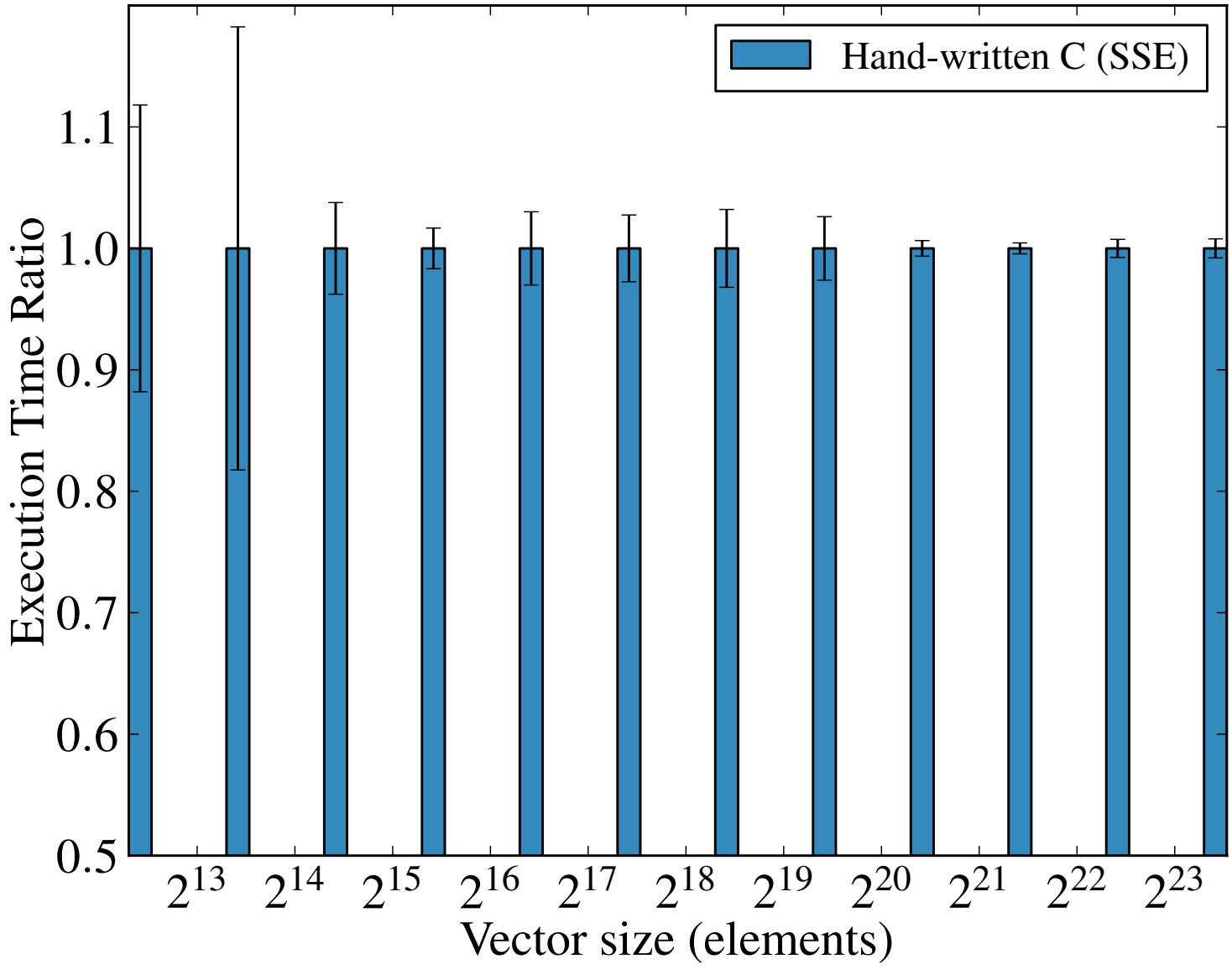
SIMD dot product in Haskell

```
dotp :: Vector Double → Vector Double → Double
dotp v w = msum (mzipWith (*) v w)
```

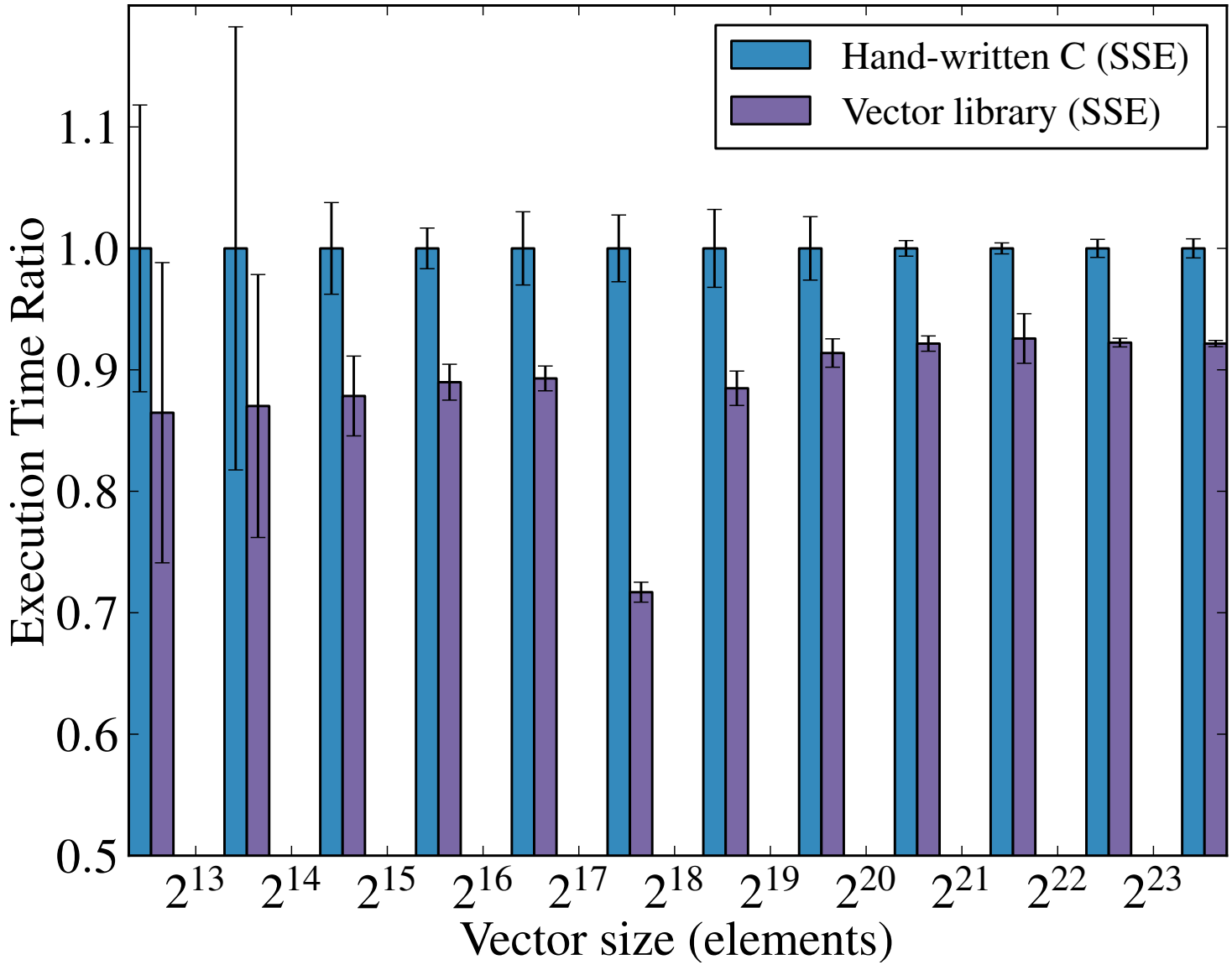
SIMD dot product in Haskell

```
dotp :: Vector Double → Vector Double → Double
dotp v w = msum (mzipWith (*) v w)

mfold' :: (Num a, Num (Multi a))
       ⇒ (∀b.Num b ⇒ b → b → b)
       → a → Vector a → a

mzipWith :: (Num a, Num (Multi a))
         ⇒ (∀b.Num b ⇒ b → b → b)
         → Vector a → Vector a → Vector a

msum :: (Num a, Num (Multi a)) ⇒ Vector a → a
msum = mfold' (+) 0
```

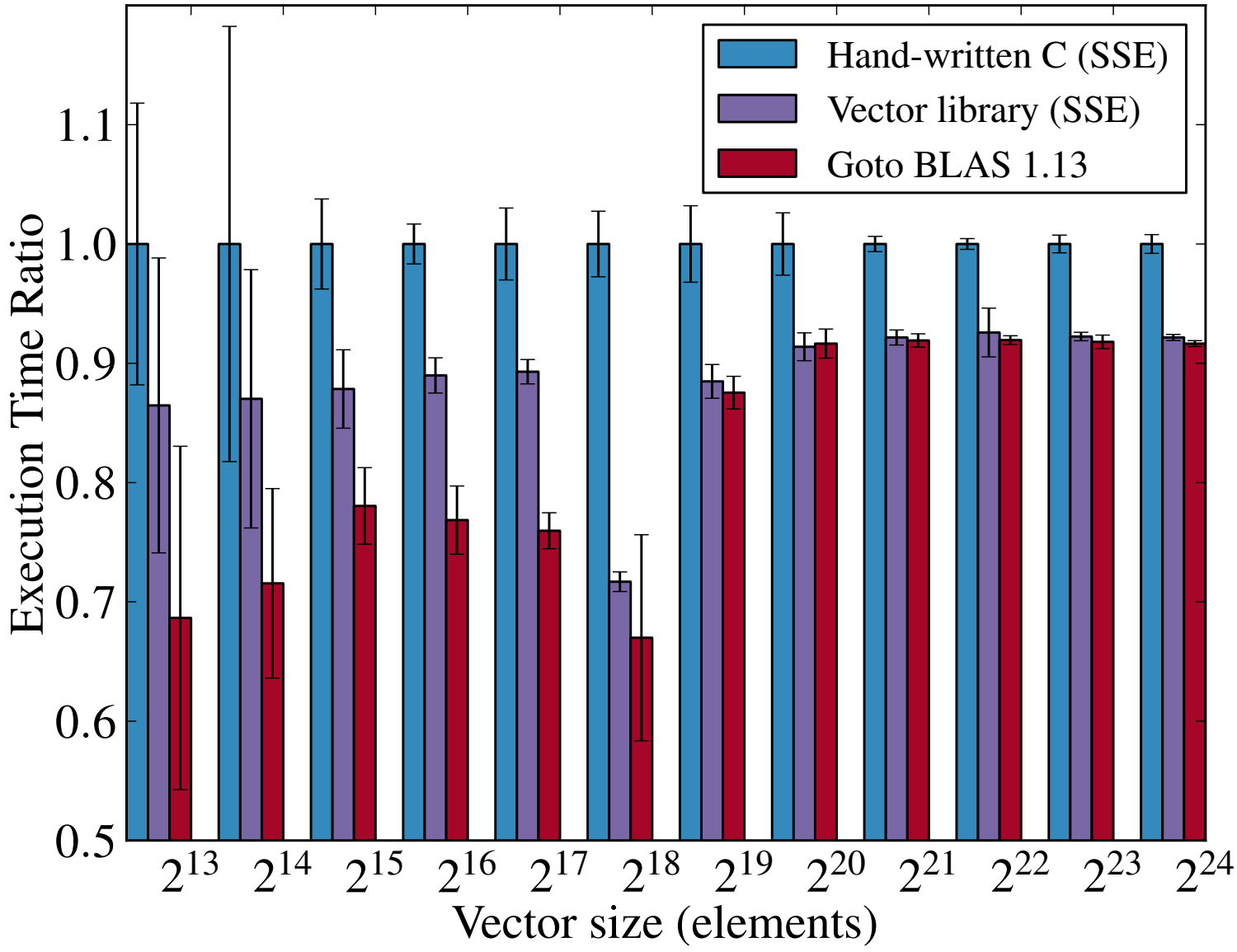
dotp performance



dotp performance: Haskell beats C



dotp performance: Haskell beats C



Haskell inner loop

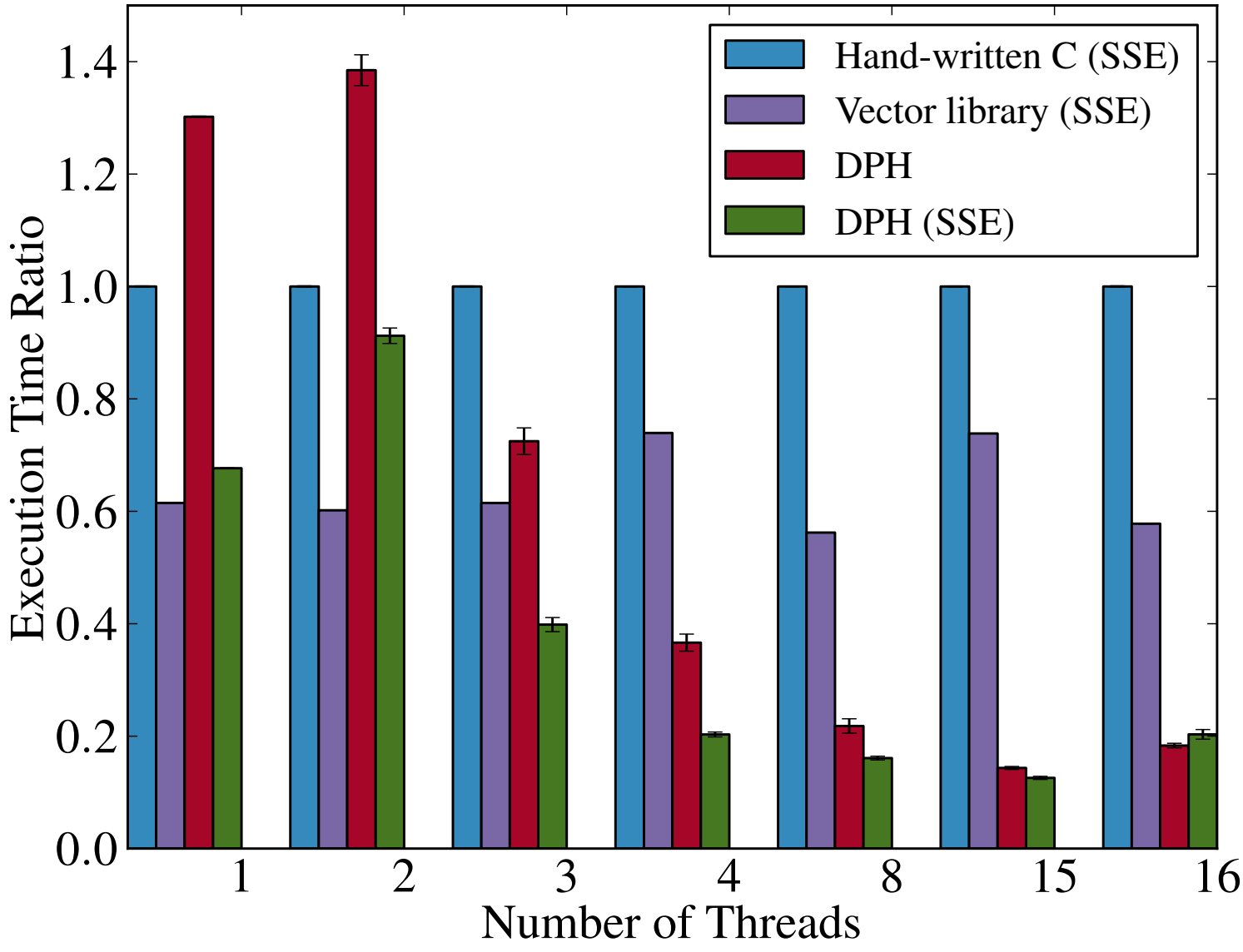
```
dotp :: Vector Double → Vector Double → Double
dotp v w = msum (mzipWith (*) v w)
```

Haskell inner loop

$$\text{dotp} :: \text{Vector Double} \to \text{Vector Double} \to \text{Double}$$
$$\text{dotp v w} = \text{msum } (\text{mzipWith } (*) \text{ v w})$$

```
.LBB5_5:
    prefetcht0 (%rdx)
    movupd     -1408(%rdx), %xmm2
    prefetcht0 (%rsi)
    movupd     -1408(%rsi), %xmm1
    mulpd      %xmm2, %xmm1
    addpd      %xmm1, %xmm0
    addq       $16, %rdx
    addq       $16, %rsi
    addq       $2, %rax
    cmpq       %rcx, %rax
    jl         .LBB5_5
```

Parallelization for free



Abstraction without cost: radial basis function

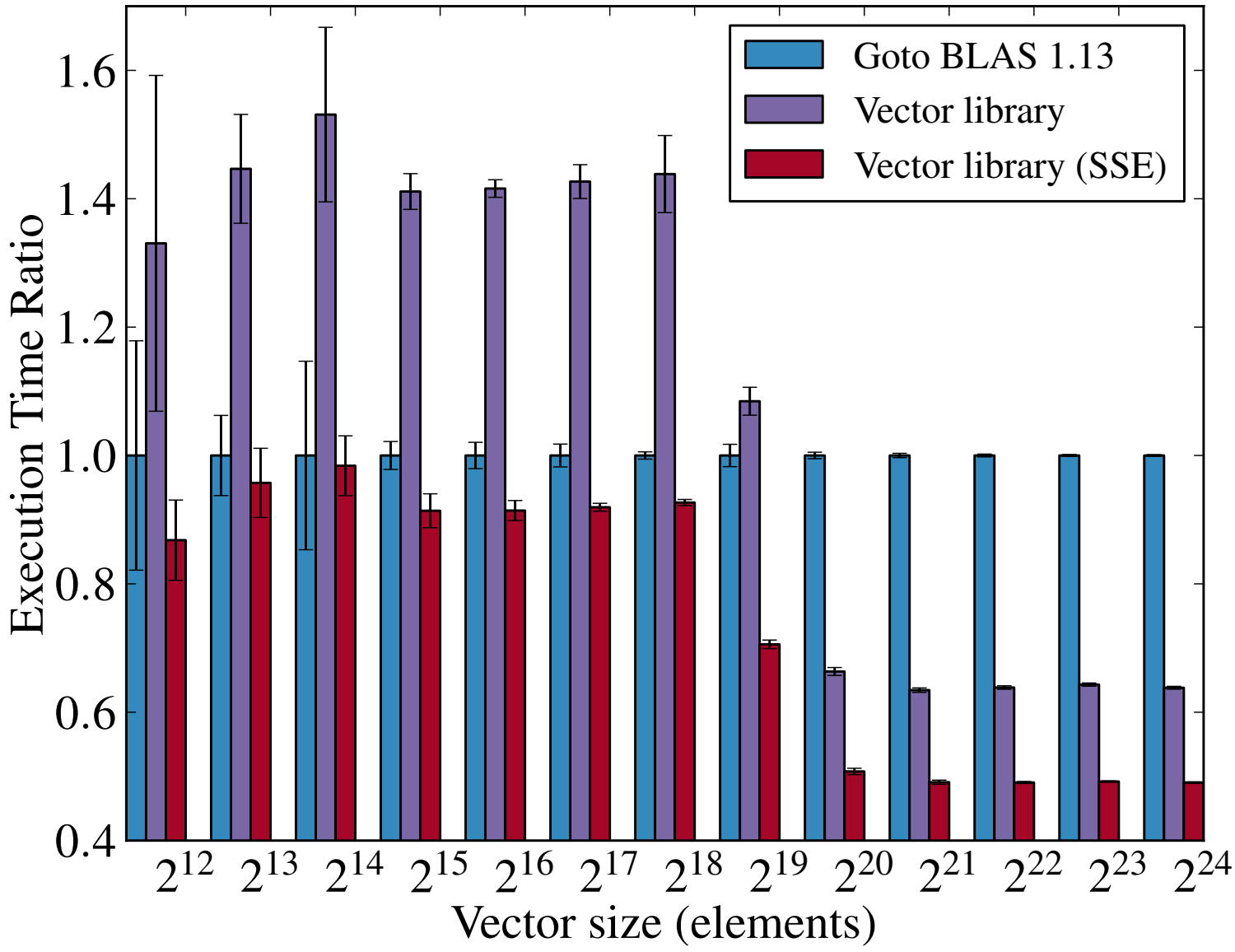
$$K(\vec{x}, \vec{y}) = e^{-\nu \|\vec{x} - \vec{y}\|^2}$$

Abstraction without cost: radial basis function

$$K(\vec{x}, \vec{y}) = e^{-\nu \|\vec{x} - \vec{y}\|^2}$$

```
rbf :: Double → Vector Double → Vector Double → Double
rbf ν v w = exp (−ν ∗ msum (mzipWith norm v w))
  where
    norm x y = square (x − y)
    square x = x ∗ x
```

rbf performance



Generalized Stream Fusion Conclusions

- Key idea: simultaneously maintain multiple representations with different cost models.
- Can make use of bulk memory operations like `memcpy` and `memset` (in draft paper).
- Allows us to take advantage of SSE instructions from high-level code.
- Use of SSE instructions in DPH requires no code modifications.
- Not limited to Haskell.