Haskell Beats C Using 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

Abstract

Stream fusion [6] is a powerful technique for automatically transforming high-level sequence-processing functions into efficient implementations. It has been used to great effect in Haskell libraries for manipulating byte arrays, Unicode text, and unboxed vectors. However, some operations, like vector append, still do not perform well within the standard stream fusion framework. Others, like SIMD computation using the SSE and AVX instructions available on modern x86 chips, do not seem to fit in the framework at all.

In this paper we introduce generalized stream fusion, which solves these issues. The key insight is to *bundle* together multiple stream representations, each tuned for a particular class of stream consumer. We also describe a stream representation suited for efficient computation with SSE instructions. Our ideas are implemented in modified versions of the GHC compiler and `vector` library. Benchmarks show that high-level Haskell code written using our compiler and libraries can produce code that is faster than both compiler- and hand-vectorized C.

1. Introduction

It seems unreasonable to ask a compiler to be able to turn numeric algorithms expressed as high-level Haskell code into tight machine code. The compiler must cope with boxed numeric types, handle lazy evaluation, and eliminate intermediate data structures. However, the Glasgow Haskell Compiler has become "sufficiently smart" that Haskell libraries for expressing numerical computations, such as Repa [16, 20], no longer have to sacrifice speed at the altar of abstraction.

The key development that made this sacrifice unnecessary is *stream fusion* [6]. Algorithms over sequences—whether they are lists or vectors (arrays)—are expressed naturally in a functional language using operations such as folds, maps, and zips. Although highly modular, these operations produce unnecessary intermediate structures that lead to inefficient code. Eliminating these intermediate structures is termed deforestation, or fusion. Equational laws, such as $\mathsf{map}\ \mathsf{f} \circ \mathsf{map}\ \mathsf{g} \equiv \mathsf{map}\ (\mathsf{f} \circ \mathsf{g})$, allow some of these intermediate structures to be eliminated; finding more general rules has been the subject of a great deal of research [7, 11, 14, 22, 28, 29, 33]. Stream fusion [6], based on the observation that recursive structures can be transformed into non-recursive co-structures for which fusion is relatively straightforward, was the first truly general solution.

Instead of working directly with lists or vectors, stream fusion works by re-expressing these functions as operations over streams, each represented as a state and a step function that transforms the state while potentially yielding a single value. Alas, different operations need different stream representations, and no single representation works well for all operations (§2.3). Furthermore, for many operations it is not obvious what the choice of representation should be.

In this paper we solve this problem with a new *generalized stream fusion* framework where the primary abstraction over which operations on vectors are expressed is a *bundle* of streams. The streams are chosen so that for any given high-level vector operation there is a stream in the bundle whose representation leads to an efficient implementation. The bundle abstraction has no run-time cost because standard compiler optimizations eliminate intermediate bundle structures. In addition, we describe several optimized stream representations. Our contributions are as follows:

- We describe a generalization of stream fusion that bundles together multiple, alternate representations of a stream. This allows the stream consumer to choose the representation that is most efficient for its use case (Section 3.1).
- We show in Section 3.2 how generalized stream fusion enables the use of bulk memory operations, such as `memcpy` and `memset`, for manipulating vectors.
- Generalized stream fusion opens up entirely new opportunities for optimization. We exploit this opportunity in Section 3.3 by crafting a stream representation well-suited for SIMD-style vector operations (SSE, AVX etc). The result is a massive performance gain (up to a factor of more than three) at a much lower programming cost than using conventional approaches (Section 5).
- Using generalized stream fusion, we describe in Section 3.5 how to modify Data Parallel Haskell [4] so that DPH programs can automatically take advantage of SIMD operations without requiring any code modification.

Everything we describe is fully implemented in GHC and its libraries (Section 4). Our benchmarks compare to the very best C and C++ compilers and libraries that we could find. Remarkably, our benchmarks show that choosing the proper stream representations can result in machine code that beats compiler-vectorized C, and that is competitive with hand-tuned assembly. Moreover, using DPH, programs can easily exploit SIMD instructions *and* automatically parallelize to take advantage of multiple cores.

2. Stream Fusion Background

We begin by providing the background necessary for understanding stream fusion. There is no new material here—it is all derived from Coutts et al. [6]. However, we describe fusion for functions of *vectors* of unboxed values, as implemented in the `vector` [18] library, rather than fusion for functions over *lists*. Some of the implementation details are elided, but the essential aspects of stream fusion as we describe them are faithful to the implementation.

2.1 The key insight

The big idea behind stream fusion is to rewrite recursive functions, which are difficult for a compiler to automatically optimize, as non-recursive functions. The abstraction that accomplishes this is the Stream data type:

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

A stream is a triple of values: an existentially-quantified state, represented by the type variable s in the above definition, a size, and a step function that, when given a state, produces a Step. A Step may be Done, indicating that there are no more values in the Stream, it may Yield a value and a new state, or it may produce a new state but Skip producing a value. The presence of Skip allows us to easily express functions like filter within the stream fusion framework.

To see concretely how this helps us avoid recursive functions, let us write map for vectors using streams

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

The functions stream and unstream convert a Vector to and from a stream. A Vector is converted to a stream whose state is an integer index and whose step function yields the value at the current index, which is incremented at each step. To convert a stream back into a Vector, unstream allocates memory for a new vector and writes each element to the vector as it is yielded by the stream—unstream embodies a recursive loop. Though imperative, the allocation and writing of the vector are safely embedded in pure Haskell using the ST monad [17].

The real work is done by map_s, which is happily non-recursive.

```
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       → Done
```

With this definition, the equational rule mentioned in the Introduction, map f ∘ map g ≡ map (f ∘ g), falls out automatically. To see this, let us first inline our new definition of map in the expression map f ∘ map g.

```
map f ∘ map g ≡
  unstream ∘ map_s f ∘ stream ∘ unstream ∘ map_s g ∘ stream
```

Given this form, we can immediately spot where an intermediate structure is formed—by the composition stream ∘ unstream. We can also see that this composition is the identity function, so we should be able to eliminate it entirely! Rewrite rules [27] enable programmers to express algebraic identities such as stream ∘ unstream = id in a form that GHC can understand and automatically apply. Stream fusion relies *critically* on this ability, and the `vector` library includes exactly this rule. With the rule in place, GHC transforms our original composition of maps into

```
map f ∘ map g ≡
  unstream ∘ map_s f ∘ map_s g ∘ stream
```

Conceptually, stream fusion pushes all recursive loops into the final consumer. The two composed invocations of map become a composition of two *non-recursive* calls to map_s. The inliner is now perfectly capable of combining map_s f ∘ map_s g into a single Stream function. Stream fusion gives us the equational rule map f ∘ map g ≡ map (f ∘ g) *for free*.

2.2 Fusing the vector dot product

The motivating example we will use for the rest of the paper is the vector dot product. A high-level implementation of this function in Haskell might be written as follows:

```
dotp :: Vector Double → Vector Double → Double
dotp v w = sum (zipWith (∗) v w)
```

It seems that this implementation will suffer from severe inefficiency—the call to zipWith produces an unnecessary intermediate vector that is immediately consumed by the function sum. In expressing dotp as a composition of collective operations, we have perhaps gained a bit of algorithmic clarity but in turn we have incurred a performance hit.

We have already seen how stream fusion eliminates intermediate structures in the case of a composition of two calls to map. Previous fusion frameworks could handle that example, but were stymied by the presence of a zipWith. However, stream fusion has no problem fusing zipWith, which we can see by applying the transformations we saw in Section 2.1 to dotp.

The first step is to re-express each Vector operation as the composition of a Stream operation and appropriate conversions between Vectors and Streams at the boundaries. The functions zipWith and sum are expressed in this form as follows.

```
zipWith :: (a → b → c) → Vector a → Vector b → Vector c
zipWith f v w = unstream (zipWith_s f (stream v) (stream w))
sum :: Num a ⇒ Vector a → a
sum v = foldl′_s 0 (+) (stream v)
```

It is now relatively straightforward to transform dotp to eliminate the intermediate structure.

```
dotp :: Vector Double → Vector Double → Double
dotp ≡ sum (zipWith (∗) v w)
     ≡ foldl′_s 0 (+) (stream (unstream
         (zipWith_s (+) (stream v) (stream w))))
     ≡ foldl′_s 0 (+)
         (zipWith_s (+) (stream v) (stream w))
```

This transformation again consists of inlining a few definitions, something that GHC can easily perform, and rewriting the composition stream ∘ unstream to the identity function. After this transformation, the production (by zipWith) and following consumption (by sum) of an intermediate Vector becomes the composition of non-recursive functions on streams.

We can see how iteration is once again pushed into the final consumer by looking at the implementations of $\mathsf{foldl'}_s$ and $\mathsf{zipWith}_s$. The final consumer in dotp is $\mathsf{foldl'}_s$, which is implemented by an explicit loop that consumes stream values and combines the yielded values with the accumulator z using the function f (the call to seq guarantees that the accumulator is strictly evaluated).

```
foldl'_s :: (a → b → a) → a → Stream b → a
foldl'_s 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
```

However, in $\mathsf{zipWith_s}$ there is no loop—the two input streams are consumed until either both streams yield a value, in which case a value is yielded on the output stream, or until one of the input streams is done producing values. The internal state of the stream associated with $\mathsf{zipWith_s}$ contains the state of the two input streams and a one-item buffer for the value produced by the first input stream.

```
zipWith_s :: (a → b → c) → Stream a → Stream b → Stream c
zipWith_s 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
```

Given these definitions, call-pattern specialization [23] in concert with the standard GHC inliner suffice to transform dotp into a single loop that does not produce an intermediate structure, and the moral result is shown here, where !! is infix byte array indexing.

```
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
```

If there is any doubt that this results in efficient machine code, we give the actual assembly language inner loop output by GHC using the LLVM back end [30]. Stream fusion preserves the ability to write compositionally without sacrificing performance.

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

2.3 Stream fusion inefficiencies

Unfortunately, while stream fusion does well for the examples we have shown, it still does not produce efficient implementations for many other operations one might want to perform on vectors. We next give examples of two classes of operations that are either inefficient or impossible to handle in the stream fusion framework. As we describe in Section 3, generalized stream fusion can implement both of these classes of operations efficiently.

2.3.1 Bulk memory operations

Appending two vectors is a simple operation for which an obvious, efficient implementation exists—a vector large enough to hold the result is allocated, and the two vectors being appended are copied, one after the other, to the destination using an optimized `memcpy`. Similarly, replicating a scalar across a vector should be implemented by a call to `memset`. In other words, we would like to be able to take advantage of highly optimized bulk memory operations like `memcpy` and `memset` in our vector library. Unfortunately, this is far from straightforward. To see why, let us look at how we might implement vector append using stream fusion.

In this framework, vector append is first rewritten in terms of stream, unstream, and a worker function $\mathsf{append_s}$.

```
append :: Vector a → Vector a → Vector a
append u v = unstream (append_s (stream u) (stream v))
append_s :: Stream a → Stream a → Stream a
append_s (Stream stepa sa na) (Stream stepb sb nb) =
    Stream step (Left sa) (na + nb)
  where
    step (Left sa) =
      case stepa sa of
        Yield x sa' → Yield x (Left sa')
        Skip sa'    → Skip    (Left sa')
        Done        → Skip    (Right sb)
    step (Right sb) =
      case stepb sb of
        Yield x sb' → Yield x (Right sb')
        Skip sb'    → Skip    (Right sb')
        Done        → Done
```

The function $\mathsf{append_s}$ appends two streams by first yielding all the values produced by the first stream and then those values produced by the second stream. When passed to unstream, this results in two loops, one executed after the other, that write elements one by one into a newly allocated vector. Similarly, replication compiles to a loop that writes, one by one, a constant to each element of a newly allocated vector. While it is theoretically possible for a compiler to transform this into code equivalent to `memcpy` and `memset`, this requires significant low-level optimization capabilities and is currently beyond the reach of GHC.

How could we possibly rewrite $\mathsf{append_s}$ into two calls to `memcpy`? We cannot. The difficulty is that a Stream produces a single value at a time. To take advantage of `memcpy`, it seems we need a stream that produces *entire vectors*, one at a time. However, this latter representation, though ideal for vector append, would be a dismal failure if we wished to calculate a dot product!

2.3.2 SIMD computation with vectors

The inadequacy of the single-value-at-a-time nature of streams becomes even more apparent when attempting to opportunistically utilize the SIMD instructions available on many current architectures, e.g., SSE on x86 and NEON on ARM. These instructions operate in parallel on data values that contains two (or four or eight, depending on the hardware architecture) floating point numbers. To avoid notational confusion, we call these *multi-values*, or sometimes just *multis*.

To enable sum to use SIMD instructions, we would like a stream representation that yields multi-values (rather than scalars), with perhaps a bit of scalar "dribble" at the end of the stream when the number of scalar values is not divisible by the size of a multi.

A stream of scalar values is useless for SIMD computation. However, a stream of multi-values isn't quite right either, because of the "dribble" problem. Perhaps we could get away with a stream

```
newMVector    :: Int → ST s (MutableVector s a)
sliceMVector  :: MutableVector s a → Int → Int
              → MutableVector s a
readMVector   :: MutableVector s a → Int → ST s a
writeMVector  :: MutableVector s a → Int → a → ST s ()
copyMVector   :: Vector a → MutableVector s a → ST s ()
freezeMVector :: MutableVector s a → ST s (Vector a)
```

Figure 1: **Operations on mutable vectors.**

that yielded *either* a scalar or a multi at each step, but this would force all scalar-only operations to handle an extra case, complicating the implementations of *all* operations and making them less efficient. There is a better way!

3. Generalized Stream Fusion

We saw in the previous Section that different stream operations work best with different stream representations. In this Section, we describe new stream representations that take advantage of bulk memory operations (§3.2) and enable SIMD computation with vectors (§3.3). Finally, we show how to use our framework to transparently take advantage of SIMD instructions in Data Parallel Haskell programs (§3.5).

3.1 Bundles of streams

The idea underlying generalized stream fusion is straightforward but its effects are wide-ranging: instead of transforming a function over vectors into a function over streams, transform it into a function over a *bundle* of streams. A bundle is simply a collection of streams, each semantically identical but with a different cost model. Individual stream operations can therefore choose to work with the most advantageous stream representation in the bundle. We give a simplified version of the Bundle data type here.

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

The sElems field of the Bundle data type contains the familiar stream of scalar values that we saw in Section 2. In Section 3.3 we describe the representation contained in the sMultis field of the record, which enables the efficient use of SSE instructions. As we show next, the stream of Chunks contained in the sChunks field of the record enables the use of bulk memory operations.

3.2 Taking advantage of bulk memory operations with Chunks

We observed in Section 2.3.1 that append could take advantage of `memcpy` if a stream could produce whole vectors rather than single scalar values in one step. The mechanism we use is slightly more general: a Chunk is a computation that destructively initializes a vector slice of a particular length:

```
data Chunk a = Chunk Int (∀s.MutableVector s a → ST s ())
```

A MutableVector s a is the mutable cousin of a Vector a. The extra type variable s is the state token that allows us to safely embed imperative code in pure computations using the ST monad [17]. Some of the operations on mutable vectors are given in Figure 1. These include operations such as copyMVector which ultimately uses a variant of `memcpy`. A producer can use these to generate Chunks which initialize large parts of a vector in one step:

```
stream :: Vector a → Bundle a
stream v = Bundle {...sChunks = singleton chunk...}
  where
    chunk = Chunk (length v) (copyMVector v)
```

Here, stream produces only one Chunk which copies the entire vector in one efficient bulk operation. To convert a Bundle to a vector, we now simply allocate a single mutable vector of sufficient size and initialize it by applying each Chunk to the appropriate slice:

```
unstream :: Bundle a → Vector a
unstream (Bundle {sChunks = Stream step s n}) =
    runST $ do mvec ← newMVector n
               loop mvec 0 s
  where
    loop mvec i s =
      case step s of
        Yield (Chunk k fill) s′ → do
          fill (sliceMVector mvec i k)
          loop mvec (i + k) s′
        Skip s′ → loop mvec i s′
        Done   → freezeMVector mvec
```

This representation is ideal for both append and replicate. The former appends the sChunks components of the two bundles using the version of $\mathsf{append_s}$ from Section 2.3.1. When unstream uses the resulting stream of chunks to manifest the vector, it automatically takes advantage of bulk memory operations. Similarly to stream, replicate produces only one Chunk that ultimately invokes `memset`. Thus, if we append a vector produced by stream and a vector produced by replicate, we will perform one `memcpy` followed by one `memset` which is as efficient as possible.

Though ideal for appending vectors efficiently, streams of chunks are not a useful representation for operations like zipWith or fold. This is not a problem, however, since the stream-of-values representation is still available in the sElems component of the Bundle. In general, each consumer will choose the best representation for the particular operation.

What happens when we append two streams that resulted from separate $\mathsf{map_s}$ operations? There is a natural conversion from a stream of scalar values to a stream of chunks—each scalar value becomes a chunk that writes a single value to a mutable vector. The result of $\mathsf{map_s}$ is therefore a Bundle that contains a degenerate stream of chunks in the sChunks field. There is not much we can do about this—mapping a function over each element in the two streams precludes us from using `memcpy` to append them. Fortunately, the degenerate stream of chunks produced by $\mathsf{map_s}$ does not in the end impose any overhead thanks to the optimizer. Furthermore, if we really are appending two vectors without first altering their contents, generalized stream fusion does not force us to give up an implementation in terms of efficient memory copies in order to gain fusion for other operations, like map. While a consumer chooses *one* representation, a producer outputs a bundle containing *all* representations, though some may be degenerate. The "degenerate" form—a stream of scalar values—results in no worse an implementation that non-generalized stream fusion. In other words, generalized stream fusion always produces code that is at least as good as that produced by stream fusion.

It might seem that maintaining a bundle of streams risks work duplication since the result is computed multiple times. Crucially, the stream representations are functions which only do work when applied by a consumer such as unstream and just like unstream, all consumers pick *exactly one* representation and discard all others so no work is ever duplicated. Standard optimizations are typically able to resolve this at compile time so that no code is duplicated,

```
class MultiType a where
  data Multi a-- Associated type

  -- The number of elements of type a in a Multi a.
  multiplicity :: Multi a → Int

  -- A Multi a containing the values 0, 1, ...,
  -- multiplicity - 1.
  multienum :: Multi a

  -- Replicate a scalar across a Multi a.
  multireplicate :: a → Multi a

  -- Map a function over the elements of a Multi a.
  multimap :: (a → a) → Multi a → Multi a

  -- Fold a function over the elements of a Multi a.
  multifold :: (b → a → b) → b → Multi a → b

  -- Zip two Multi a's with a function.
  multizipWith :: (a → a → a)
                 → Multi a → Multi a → Multi a
```

Figure 2: **The MultiType type class and its associated type, Multi.**

either. The scheme does not rely on lazy evaluation and would work fine in a call-by-value language.

3.3 A stream representation fit for SIMD computation

Modifying the stream fusion framework to accommodate SIMD operations requires a more thoughtful choice of representation. However, proper SIMD support opens up the possibility of dramatically increased performance for a wide range of numerical algorithms. We focus on SIMD computations using 128-bit wide vectors and SSE instructions on x86 since that is what our current implementation supports, although the approach generalizes.

Our implementation represents SIMD values using the type family Multi. We have chosen the name to avoid confusion with the Vector type which represents arrays of arbitrary extent. In contrast, a value of type Multi a is a short vector containing a *fixed* number of elements—known as its *multiplicity*—of type a. On a given platform, Multi a has a multiplicity that is appropriate for the platform's SIMD instructions. For example, on x86, a Multi Double will have multiplicity 2 since SSE instructions operate on 128-bit wide vectors, whereas a Multi Float will have multiplicity 4. Multi is implemented as an associated type [3] in the MultiType type class; their definitions are shown in Figure 2. MultiType includes various operations over Multi values, such as replicating a scalar across a Multi and folding a function over the scalar elements of a Multi. These operations are defined in terms of new primitives we added to GHC that compile directly to SSE instructions.

Given a value of type Vector Double, how can we operate on it efficiently using SSE instructions within the generalized stream fusion framework? An obvious first attempt is to include a stream of Multi Doubles in the stream bundle. However, this representation is insufficient for a vector with an odd number of elements since we will have one Double not belonging to a Multi at the end. Let us instead try this instead: a stream that can contain *either* a scalar or a Multi. We call this stream a MultisP because the *producer* chooses what will be yielded at each step.

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

Now we can implement summation using SIMD operations. Our strategy is to use two accumulating parameters, one for the sum of the Multi values yielded by the stream and one for the sum of the scalar values. Note that $(+)$ is overloaded: we use *SIMD* $(+)$ to add summ and y, and *scalar* $(+)$ to add sum1 and x.

```
msumPs :: (Num a, Num (Multi a)) ⇒ MultisP a → a
msumPs (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
```

When the stream is done yielding values, we call the multifold member of the MultiType type class to fold the addition operator over the components of the Multi.

This implementation strategy works nicely for folds. However, if we try to implement the SIMD equivalent of zipWith_s, we hit a roadblock. A SIMD version of zipWith_s requires that at each step either *both* of its input streams yield a Multi or they *both* yield a scalar—if one were to yield a scalar while the other yielded a Multi, we would have to somehow buffer the components of the Multi. And if one stream yielded *only* scalars while the other yielded only Multis, we would be hard-pressed to cope.

Instead of a stream representation where the producer chooses what is yielded, let us instead choose a representation where the stream *consumer* is in control.

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

The idea is for a MultisC a to be able to yield either a value of type Multi a or a value of type a—the stream consumer chooses which by calling one of the two step functions. Note that the existential state is quantified over both step functions, meaning that the same state can be used to yield either a single scalar or a Multi. If there is not a full Multi available, the first step function will return Done. The remaining scalars will then be yielded by the second step function. This representation allows us to implement a SIMD version of zipWith_s (not shown here) and to slightly improve summation:

```
msumCs :: (Num a, Num (Multi a)) ⇒ MultisC a → a
msumCs (Stream mstep sstep s _) = mloop 0.0 s
  where
    mloop summ s =
      case mstep s of
        Yield x s′ → mloop (summ + x) (Left s′)
        Skip    s′ → mloop summ       (Left s′)
        Done       → sloop  summ 0.0 s

    sloop summ sum1 s =
      case sstep s of
        Yield x s′ → sloop summ (sum1 + x) s′
        Skip    s′ → sloop summ sum1       s′
        Done       → multifold (+) sum1 summ
```

Regrettably, a MultisC *still* isn't quite what we need. Consider appending two vectors of Doubles, each of which contains 41 elements. We cannot assume that the two vectors are laid out consecutively in memory, so even though the stream that results from appending them together will contain 82 scalars, this stream is forced to yield a scalar in the middle of the stream. One might imagine an implementation that buffers and shifts partial Multi values, but this leads to very inefficient code. The alternative is for

$\mathsf{append_s}$ to produce a stream in which either a scalar or a Multi is yielded at each step—but that was the original representation we selected and then discarded because it was not suitable for zips!

The final compromise is to allow either—but not both—of these two representations. We cannot allow both—hence there is only one new bundle member rather than two—because while we can easily convert a MultisC a into a MultisP a, the other direction is not efficiently implementable. The final definition of the Multis type alias is therefore

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

Each stream function that can operate on Multi values consumes the Multis a in the sMultis field of the stream bundle. It must be prepared to accept either a MultisC or a "mixed" stream of scalars and Multi's, as this final definition of $\mathsf{msum_s}$ shows:

$$\begin{array}{l} \mathsf{msum_s :: (Num\ a, Num\ (Multi\ a)) \Rightarrow Bundle\ a \rightarrow a} \\ \mathsf{msum_s\ (Bundle\ \{sMultis = Left\ \ \ s\}) = msumC_s\ s} \\ \mathsf{msum_s\ (Bundle\ \{sMultis = Right\ t\}) = msumP_s\ t} \end{array}$$

However, we always try to produce a MultisC and only fall back to a MultisP as a last resort. Even operations that can work with either representation are often worht specializing for the MultisC form. In the case of $\mathsf{msum_s}$ above, this allows us to gobble up as many Multi values as possible and only then switch to consuming scalars, thereby cutting the number of accumulating parameters in half and reducing register pressure.

3.4 A SIMD version of dotp

With a stream representation for SIMD computation in hand, we can write a SIMD-ized version of the dot product from Section 2.

```
dotp_simd :: Vector Double → Vector Double → Double
dotp_simd v w = msum (mzipWith (∗) v w)
```

The only difference with respect to the scalar implementation in Section 2.2 is that we use variants of foldl′ and zipWith specialized to take function arguments that operate on values that are members of the Num type class. While we could have used versions of these functions that take two function arguments (our library contains both varietals), one for scalars and one for Multis, the forms that use overloading to allow the function argument to be used at both the type a → a → a and Multi a → Multi a → Multi a are a convenient shorthand.

```
mfold′ :: (PackedVector Vector a, Num a, Num (Multi a))
        ⇒ (∀b.Num b ⇒ b → b → b)
        → a → Vector a → a
mzipWith :: (PackedVector Vector a, Num a, Num (Multi a))
          ⇒ (∀b.Num b ⇒ b → b → b)
          → Vector a → Vector a → Vector a
msum :: (PackedVector Vector a, Num a, Num (Multi a))
      ⇒ Vector a → a
msum = mfold′ (+) 0
```

The particular fold we use here, mfold′, maintains two accumulators (a scalar and a Multi) when given a MultisP a, and one accumulator when given a MultisC a. The initial value of the scalar accumulator is the third argument to mfold′, and the initial value of the Multi accumulator is formed by replicating this scalar argument across a Multi. The result of the fold is computed by combining the elements of the Multi accumulator and the scalar accumulator using the function multifold, just as our implementation of $\mathsf{msum_s}$. Note that the first argument to mfold′ must be associative and commutative. The PackedVector type class constraint on mfold′, mzipWith, and msum ensures that the type a is an instance of MultiType and that elements contained in the vector can be extracted a Multi a at a time.

The stream version of mfold′, $\mathsf{mfold'_s}$, can generate efficient code no matter what representation is contained in a Multis a. On the other hand, the stream version of mzipWith, $\mathsf{mzipWith_s}$, requires that both its vector arguments have a MultisC representation. Since there is no good way to zip two streams when one yields a scalar and the other a Multi, if either bundle argument to $\mathsf{mzipWith_s}$ does not have a MultisC representation available, $\mathsf{mzipWith_s}$ falls back to an implementation that uses only scalar operations.

3.5 Automatically parallelizing SIMD computations

Using SIMD instructions does not come entirely for free. Consider mapping over a vector represented using multis:

```
mmap :: (PackedVector Vector a)
      ⇒ (a → a)
      → (Multi a → Multi a)
      → Vector a → Vector a
```

To map efficiently over the vector it does not suffice to pass a function of type (a → a), because that does not work over multis. We must also pass a semantically equivalent multi-version of the function. For simple arithmetic, matters are not too bad (it just looks stupid):

```
foo :: Vector Float → Vector Float
foo v = mmap (λx y → x + y ∗ 2) (λx y → x + y ∗ 2) v
```

The two lambdas are at different types, but Haskell's overloading takes care of that. We could attempt to abstract this pattern like this:

```
mmap :: (PackedVector Vector a)
      ⇒ (∀a.Num a ⇒ a → a)
      → Vector a → Vector a
```

But that attempt fails if you want operations in class Floating, say, rather than Num. What we want is a way to *automatically multi-ize scalar functions* (such as (λx y → x + y ∗ 2) above), so that we get a pair of a scalar function and a multi function, which in turn can be passed to map.

There is another problem: mmap only takes a function of type (a → a) which is less general (and hence less useful) than usual. The reason for this is the limited range of functions that that the hardware offers over Multis, which is reflected in the type of multimap in the MultiType class (Figure 2).

Happily, both problems are already solved by the vectorization transformation of Data Parallel Haskell [4, 24], itself based on the pioneering NESL [1]. Given a function f over scalars, DPH produces a function f′ which operates on *vectors* of scalars and is equivalent to map f, but uses only primitive collective operations rather than iteration. Of course, a Multi is a special case of a vector and MultiType includes enough functionality to provide the primitive collective operations that DPH needs which means we can use it to produce SIMD code. Better still, DPH allows us to take advantage of *multiple cores*, as well as the SIMD instruction in each core. There are many moving parts here, but in principle we can get the convenience of scalar `vector` code, and *automatically* exploit both SIMD instructions and multi-core.

We start with an advantage: DPH is already built on the stream abstraction provided by the `vector` library. We modified the DPH libraries to use our bundle abstraction instead. Because DPH programs are vectorized by the compiler so that all scalar operations are turned into operations over wide vectors, by implementing these wide vector operations using our new SIMD functions like msum, programs written using DPH automatically and transparently take advantage of SSE instructions—no code changes are required of the programmer.

3.6 How general is generalized stream fusion?

Generalized stream fusion provides a representation and algebraic laws for rewriting operations over this representation whose usefulness extends beyond Haskell. Although we have implemented generalized stream fusion as a library, it could also be incorporated into a compiler as an intermediate language. This was not necessary in our implementation because GHC's generic optimizer is powerful enough to eliminate all intermediate structures created by generalized stream fusion. In other words, GHC is such a fantastic partial evaluator that we were able to build generalized stream fusion as a library rather than incorporating it into the compiler itself. Writing high-level code without paying an abstraction tax is desirable in any language, and compilers less capable than GHC can also avoid this tax using the ideas we outline in this paper, although perhaps only by paying a substantial one-time implementation cost.

4. Implementation

There are three substantial components of our implementation. We first modified GHC itself to add support for SSE instructions. This required changing GHC's register allocator to allow overlapping register classes. Previously, single- and double-precision registers could only be drawn from disjoint sets of registers even though on many platforms, including x86 (when using SSE) and x86-64, there is a single register class for both single- and double-precision floating point values. This change was also necessary to allow SSE vectors to be stored in registers. We then added support for primitive SIMD vector types and primitive operations over these types to GHC's dialect of Haskell. These primitives are fully unboxed [25]. The STG [21] and C-- [26] intermediate languages, as well as the LLVM code generator [30], were also extended to support compiling the new Haskell SIMD primitives. Boxed wrappers for the unboxed primitives and the MultiType type class and its associated Multi type complete the high-level support for working directly with basic SIMD data types. Because the SIMD support we added to GHC utilizes the LLVM back-end, it should be relatively straightforward to adapt our modifications for other CPU architectures, although at this time only x86-64 is supported.

Second, we implemented generalized stream fusion in a modified version of the `vector` library [18] for computing with efficient unboxed vectors in Haskell. We replaced the existing stream fusion implementation with an implementation that uses the Bundle representation and extended the existing API with functions such as mfold′ and mzipWith that enable using SIMD operations on the contents of vectors. The examples in this paper are somewhat simplified from the actual implementations. For example, the actual implementations are written in monadic form and involve type class constraints that we have elided. Vectors whose scalar elements can be accessed in SIMD-sized groups, i.e., vectors whose scalar elements are laid out consecutively in memory, are actually represented using a PackedVector type class. These details do not affect the essential design choices we have described, and the functions used in all examples are simply type-specialized instances of the true implementations.

Third, we modified the DPH libraries to take advantage of our new `vector` library. The DPH libraries are built on top of the stream representation from a previous version of the `vector` library, so we first updated DPH to use our bundle representation instead. We next re-implemented the primitive wide-vector operations in DPH in terms of our new SIMD operations on bundles. While we only provided SIMD implementation for operations on double-precision floating point values, this part of the implementation was quite small, consisting of approximately 20 lines of code not counting `ifdef`s. Further extending SIMD support in DPH will be easy now that it is based on bundles rather than streams.

```
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;
}
```

Figure 3: **C implementation of vector dot product using only scalar operations.**

Our register allocation patch and our SIMD patches supporting SSE instructions have already been accepted into mainline GHC. Our modifications to the `vector` and DPH libraries are available in a public git repository. We expect these library modifications to also be adopted by their respective maintainers. We are working with all maintainers to ensure that the work we report on in this paper will be available in the GHC 7.8 release along with support for AVX instructions.

5. Evaluation

Our original goal in modifying GHC and the `vector` library was to make efficient use of SSE instructions from high-level Haskell code. The inability to use SSE operations from Haskell and its impact on performance is a deficiency that was brought to our attention by Lippmeier and Keller [19]. The first step we took was to write a small number of simple C functions utilizing SSE intrinsics to serve as benchmarks. This gave us a very concrete goal—to generate machine code from Haskell that was competitive with these C implementations.

For comparison, we also implemented a C version of `ddotp` using only scalar operations, shown in Figure 3. The C implementation we use as a benchmark for double-precision dot product, to which we have added support for SSE by hand, is given in Figure 4. We repeat the definition of the Haskell implementation here.

$$\mathsf{ddotp} :: \mathsf{Vector\ Double} \to \mathsf{Vector\ Double} \to \mathsf{Double}$$
$$\mathsf{ddotp\ v\ w} = \mathsf{mfold}'\ (+)\ 0\ (\mathsf{mzipWith}\ (*)\ \mathsf{v\ w})$$

Though not exactly onerous, the C version with SSE support is already unpleasantly more complex than the scalar version. And surely the Haskell version, consisting of a single line of code (not including the optional type signature) is a good bit simpler. Also note that the Haskell programmer can think compositionally—it is natural to think of dot product as pair-wise multiplication followed by summation. The C programmer, on the other hand, must manually fuse the two loops into a single multiply-add. Furthermore, as well as being constructed compositionally, the Haskell implementation can itself be *used* compositionally. That is, if the input vectors to ddotp are themselves the results of vector computations, generalized stream fusion will potentially fuse all operations in the chain into a single loop. In contrast, the C programmer must manifest the input to the C implementation of `ddotp` as concrete vectors in memory—there is no potential for automatic fusion with other operations.

Figure 5 compares the single-threaded performance of several implementations of the dot product, including C and Haskell versions that only use scalar operations as well as the implementation provided by GotoBLAS2 1.13 [8, 9]. Times were measured

```
#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;
}
```

Figure 4: **C implementation of vector dot product using SSE intrinsics.**

on a 3.40GHz Intel i7-2600K processor, averaged over 100 runs. To make the relative performance of the various implementations clearer, we show the execution time of each implementation relative to the scalar C version (from Figure 4), which is normalized to 1.0, in Figure 6.

Surprisingly, both the naive scalar C implementation (Figure 3) and the version written using SSE intrinsics (Figure 4) perform approximately the same. This is because GCC in fact vectorizes the scalar implementation. However, the Haskell implementation is almost always faster than both C versions; it is 5-20% slower for very short vectors (those with fewer than about 16 elements) and 1-2% slower just when the working set size exceeds the capacity of the L1 cache. Not only does Haskell beat C, but it *beats GCC's vectorizer*! Once the working set no longer fits in L3 cache, the Haskell implementation is even neck-and-neck with the implementation of `ddotp` from GotoBLAS, a collection of highly-tuned BLAS routines hand-written in assembly language that is generally considered to be one of the fastest BLAS implementation available.

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: *loop unrolling* and *prefetching*.

The generalized stream fusion framework allowed us to implement the equivalent of loop unrolling by adding under 200 lines of code the to `vector` library. We changed the MultisC data type to incorporate a *leap*, which is a Step that contains multiple values of type Multi a. We chose Leap to contain four values—so loops are unrolled four times—since on x86-64 processors this tends not to put too much register pressure on the register allocator. Adding multiple Leaps of different sizes would also be possible. MultisC

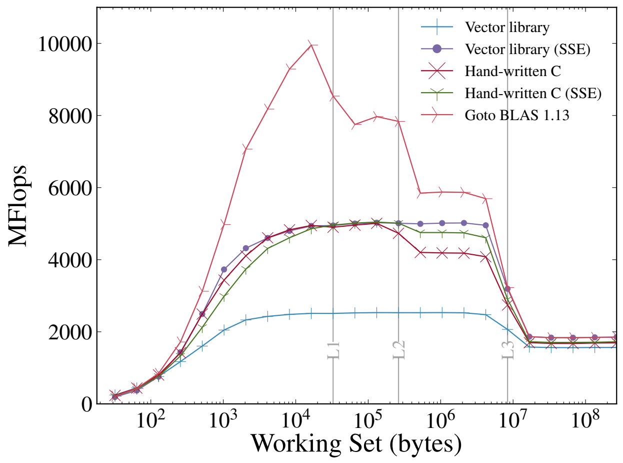

Figure 5: **Single-threaded performance of double-precision dot product implementations.** C implementations were compiled using GCC 4.7.2 and compiler options `-O3 -msse4.2 -ffast-math -ftree-vectorize -funroll-loops`. Sizes of the L1, L2, and L3 caches are marked.

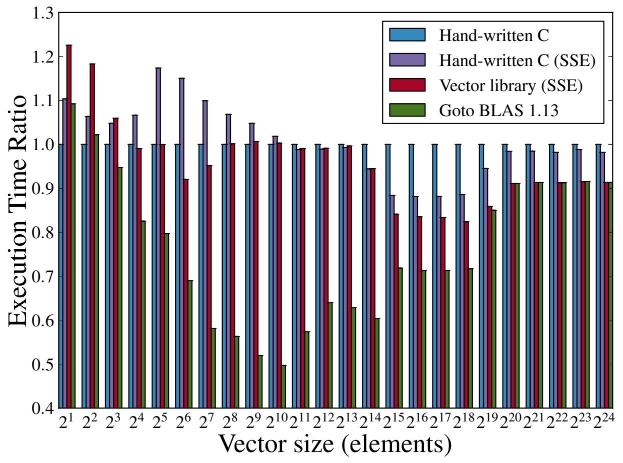

Figure 6: **Relative performance of single-threaded `ddot` implementations.** All times are normalized relative to the hand-written, compiler-vectorized, C implementation.

consumers may of course chose not to use the Leap stepping function, in which case loops will not be unrolled.

$$\textbf{data}\ \mathsf{Leap\ a} = \mathsf{Leap\ a\ a\ a\ a}$$

$$\begin{aligned}&\textbf{data}\ \mathsf{MultisC\ a}\ \textbf{where}\\ &\quad \mathsf{MultisC} :: (\mathsf{s} \to \mathsf{Step\ s\ (Leap\ (Multi\ a))})\\ &\qquad\qquad \to (\mathsf{s} \to \mathsf{Step\ s\ (Multi\ a)})\\ &\qquad\qquad \to (\mathsf{s} \to \mathsf{Step\ s\ a})\\ &\qquad\qquad \to \mathsf{s}\\ &\qquad\qquad \to \mathsf{MultisC\ a}\end{aligned}$$

Prefetch instructions on Intel processors allow the program to give the CPU a hint about memory access patterns, telling it to prefetch memory that the program plans to use in the future. In our library, these prefetch hints are implemented using prefetch primitives that we added to GHC. When converting a Vector to

```
.LBB4_12:
    prefetcht0 1600(%rcx,%rdi)
    vmovupd    64(%rcx,%rdi), %xmm1
    prefetcht0 1600(%rsi,%rdi)
    vmovupd    80(%rcx,%rdi), %xmm2
    vmulpd     80(%rsi,%rdi), %xmm2, %xmm2
    vmulpd     64(%rsi,%rdi), %xmm1, %xmm1
    vaddpd     %xmm1, %xmm0, %xmm0
    vaddpd     %xmm2, %xmm0, %xmm0
    vmovupd    96(%rcx,%rdi), %xmm2
    vmovupd    96(%rsi,%rdi), %xmm3
    vmovupd    112(%rcx,%rdi), %xmm1
    vmulpd     112(%rsi,%rdi), %xmm1, %xmm1
    vmulpd     %xmm2, %xmm3, %xmm2
    addq       $64, %rdi
    leaq       8(%rax), %rdx
    addq       $16, %rax
    vaddpd     %xmm2, %xmm0, %xmm0
    cmpq       %rbx, %rax
    vaddpd     %xmm1, %xmm0, %xmm0
    movq       %rdx, %rax
    jle        .LBB4_12
```

Figure 7: **Inner loop of Haskell** ddotp **function.**

a MultisC, we know exactly what memory access pattern will be used—each element of the vector will be accessed in linear order. The function that performs this conversion, stream, takes advantage of this knowledge by executing prefetch instructions as it yields each Leap. Only consumers using Leaps will compile to loops containing prefetch instructions, and stream will only add prefetch instructions for vectors whose size is above a fixed threshold (currently 8192 elements), because for shorter vectors the extra instruction dispatch overhead is not amortized by the increase in memory throughput. Loop unrolling and prefetching produce an inner loop for our Haskell implementation of ddotp that is shown in Figure 7.

Not only can the client of our modified vector library write programs in terms of boxed values and directly compose vector operations instead of manually fusing operations without paying an abstraction penalty, but he or she can transparently benefit from low-level prefetch magic baked into the library. Of course the same prefetch magic could be expressed manually in the C version. However, the author who wrote the code in Figure 4 did not know about prefetch at the time of implementation. We suspect that many C programmers are in the same state of ignorance. In Haskell, this knowledge is embedded in a library, and clients benefit from it automatically.

5.2 Speeding up Haskell

To see how easy it is to integrate SIMD instruction into existing programs, we rewrote a number of functions from various packages using our modified vector library. In all cases, we simply identified suitable operations (typically maps and folds) and replaced them by multi-enabled versions. We did *not* modify or refactor the algorithms. The percentage speedup of the rewritten versions is shown in Figure 8. The 'sum', 'kahan', 'dotp', 'saxpy', and 'rbf' benchmarks are short programs that make heavy use of numerics ('kahan' implements Kahan summation [15]). Two benchmarks, 'variance' and 'kde', a kernel density estimator, are adapted from the statistics [2] Haskell package. These first seven benchmarks were run using vectors containing 2^{16} Doubles. The 'mixture' benchmark is adapted from the StatisticalMethods [5] Haskell package and implements the expectation-maximization

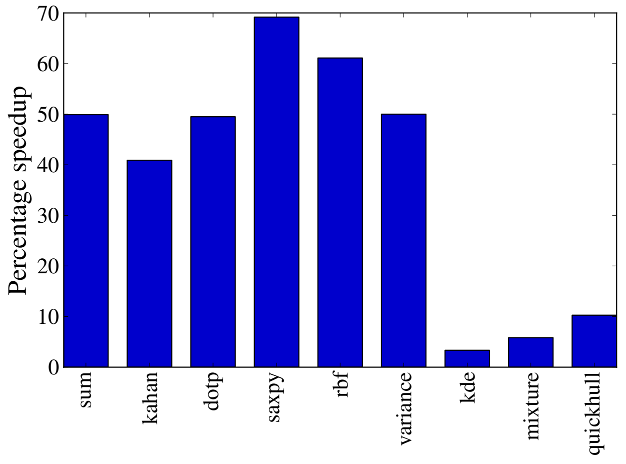

Figure 8: **Percentage improvement in running time.**

(EM) algorithm for a mixture of two Gaussians [12, §8.5.1]. The final benchmark, 'quickhull', is adapated from the examples included with DPH.

The first six benchmarks consist almost entirely of numeric operations for which SSE version are available; correspondingly, they show a significant speedup, from more than 1.5× to more than 3×. The final three functions contain features that frustrated our efforts, such as use of non-numeric operations or data-dependent control flow, which is difficult to vectorize. One benchmark, 'mixture', has a run time that is exponential in the size of its input, so 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. We expect that many Haskell programs that make some use of numerical operations on sequences will have at least a few such ripe morsels. Rewriting a small amount of code for a 5% performance improvement is worthwhile; rewriting a small amount of code for a 3× performance improvement is an unbelievably good deal.

5.3 Abstraction without cost

Using Haskell for implementing kernels such as dotp provides us with another significant advantage: these kernels can fuse with other vector operations! Fusion is not possible with BLAS routines but can dramatically increase performance. Consider the Gaussian radial basis function [12], defined as

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

A C programmer implementing this function could take advantage of high-performance BLAS routines in two possible ways. The first is to write $\mathbf{x}-\mathbf{y}$ to a temporary vector and then call BLAS. Of course this requires an intermediate structure, which is undesirable. The second possibility is to apply the identity $\|\mathbf{x}-\mathbf{y}\|^2 = \mathbf{x}\cdot\mathbf{x} - 2\mathbf{x}\cdot\mathbf{y} + \mathbf{y}\cdot\mathbf{y}$ and call into the BLAS three times without creating a temporary vector. This does not require an intermediate structure, but it requires making multiple passes over each input vector. It is also numerically unstable. The Haskell programmer, on the other hand, could straightforwardly write the following implementation

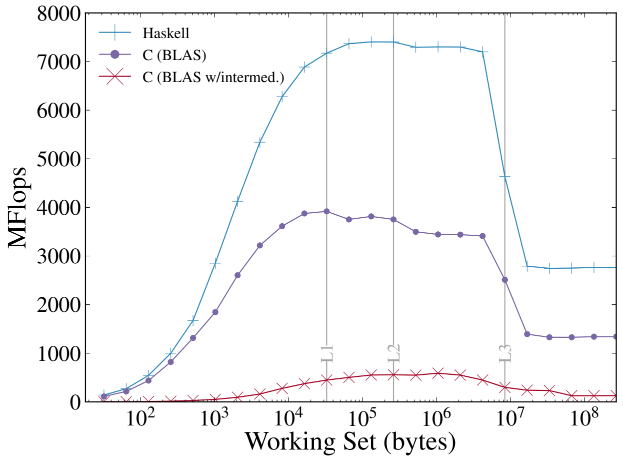

Figure 9: **Performance of Haskell and C Gaussian RBF implementations.**.

$$\begin{aligned}
&\mathsf{rbf} :: \mathsf{Double} \to \mathsf{Vector\ Double} \to \mathsf{Vector\ Double} \to \mathsf{Double}\\
&\mathsf{rbf}\ \nu\ \mathsf{v}\ \mathsf{w} = \mathsf{exp}\ (-\nu * \mathsf{msum}\ (\mathsf{mzipWith\ norm\ v\ w}))\\
&\quad \textbf{where}\\
&\quad\quad \mathsf{norm\ x\ y} = \mathsf{square}\ (\mathsf{x} - \mathsf{y})\\
&\quad\quad \mathsf{square\ x} = \mathsf{x} * \mathsf{x}
\end{aligned}$$

We show the performance of three implementations of the Gaussian RBF in Figure 9. The first is the Haskell function rbf whose definition we have just given. The second is the C version that uses the above identity to avoid temporary allocation, and the third allocates an intermediate value. The Haskell version with SSE support is the clear winner—due to fusion, it touches each element of the two vectors only once.

5.4 Haskell beats C++?

C is a straw man when fusion of array operations becomes critical for performance—we know of no C compiler that can perform this kind of optimisation. An imperative programmer would not attack these sorts of problems with C in any case, but likely turn to C++, for which there are a number of libraries tailored to matrix computation that do perform fusion. These libraries all use expression templates [31], a technique pioneered by the Blitz++ [32] library.

We have implemented C++ versions of the Gaussian RBF using three different libraries: Blitz++ 0.10 [32], Boost uBLAS 1.53 [13], and Eigen 3.1.2 [10]. We give performance numbers for all three, but our discussion focuses on Eigen, which utilizes SSE instructions and is generally considered the most performant library in this class. Figure 10 shows the Gaussian RBF implemented using Eigen. Note that we have implemented the square of the L_2-norm ourselves, a point we return to later. In this specific case the Eigen library has the squared L_2-norm in its API, but in more complicated examples there might be no such built-in provision.

After writing the code in Figure 10, we decided to actually read the Eigen documentation. We reproduce a highly relevant note here [10]:

> To summarize, the implementation of functions taking non-writable (const referenced) objects is not a big issue and does not lead to problematic situations in terms of compiling and running your program. However, a naive implementation is likely to introduce unnecessary temporary objects in your code. In order to avoid evaluating parameters into temporaries, pass them as (const) references to MatrixBase or ArrayBase (so templatize your function).

```
double norm2(VectorXd const& v)
{
    return v.dot(v);
}

double eigen_rbf(double nu, VectorXd const& u,
    VectorXd const& v)
{
    VectorXd temp = u - v;

    return exp(-nu*norm2(u-v));
}
```

Figure 10: **C++ implementation of Gaussian radial basis function.**

```
template <typename Derived>
typename Derived::Scalar norm2(const
    MatrixBase<Derived>& v)
{
    return v.dot(v);
}

double eigen_rbf(double nu, VectorXd const& u,
    VectorXd const& v)
{
    VectorXd temp = u - v;

    return exp(-nu*norm2(u-v));
}
```

Figure 11: **C++ implementation of Gaussian radial basis function.**

Having read this, we realized our mistake and modified our code, producing the version in Figure 11. Avoiding temporary allocation in Eigen and other libraries can require some care because expression templates are not vectors or matrices, but represent *computations* that, when run, compute a matrix or vector. Programmer-written abstractions must be careful to abstract over *expression templates*, and not over matrices or vectors. Figure 12 shows the performance of the Haskell and C++ implementations of the Gaussian radial basis function. Our initial version using Eigen (from Figure 10, labeled "Eigen (bad norm2)") performs quite poorly but the modified one is the fastest by far. Interestingly, once the working set no longer fits in L3 cache, the Boost version, which does not use SSE instructions, outperforms all SSE-enabled variants except the fast Eigen version. We suspect that Boost and Eigen make better use of prefetching interact better with the processor's prefetch prediction than the other libraries.

Clearly "properly"-written C++ can outperform Haskell. The challenge is in figuring out what "proper" means. A Haskell programmer can write straightforward, declarative code and expect the compiler to handle fusion. The C++ programmer must worry about the performance implications of abstraction.

Furthermore, unlike Eigen, our implementation is immature, and there are several straightforward changes that will further improve the absolute performance of Haskell code. There is room for improvement in our use of prefetching. More importantly, we currently cannot rely on memory allocated by GHC to be properly aligned for SSE move-aligned instructions, so all SSE move instructions are unaligned. Differentiating between unaligned and aligned memory will allow us to avoid the large performance hit incurred by our current implementation.

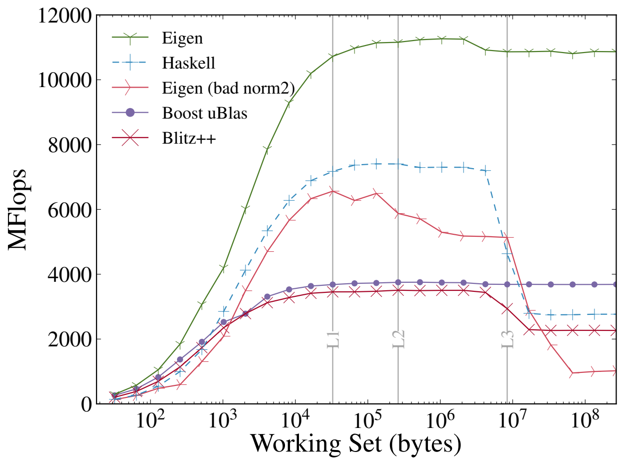

Figure 12: **Performance of Haskell and C++ Gaussian RBF implementations.**

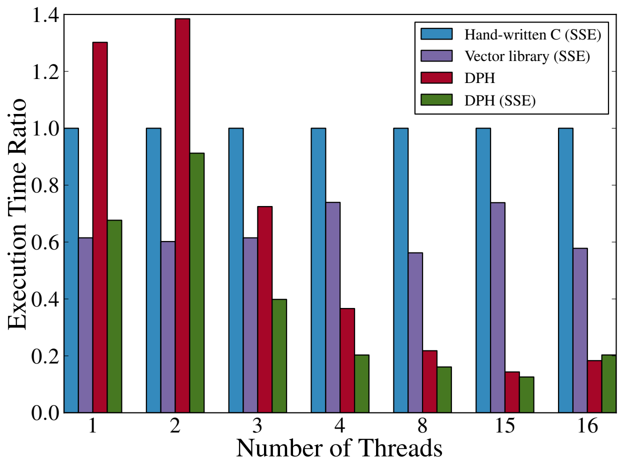

Figure 13: **Performance of double-precision dot product implementations.** The DPH implementations are multi-threaded, and the vector library and hand-written C implementations are single-threaded.

5.5 Generalized stream fusion in Data Parallel Haskell

As we described in Section 3.5, we have modified Data Parallel Haskell (DPH) to generate vectorized programs that work over bundles, and hence exploit both SSE and multi-core parallelism. Figure 13 shows the performance of several dot product implementations, including DPH implementations with and without our rewritten run-time. These measurements were taken on a 32-core (4×8) AMD Opteron 6128 running at 2GHz. Although there is some overhead due to DPH in the SSE implementation and significant overhead in the scalar implementation, DPH does scale well—in addition to transparently taking advantage of SSE instructions, our modified version of DPH transparently takes advantage of additional cores. We believe the disparity between the two DPH implementations is due in large part to non-SSE-specific optimizations we implemented.

6. Related Work

Wadler [33] introduced the problem of deforestation, that is, of eliminating intermediate structures in programs written as compositions of list transforming functions. A great deal of follow-on work [7, 11, 14, 22, 28, 29] attempted to improve the ability of compilers to automate deforestation through program transformations. Each of these approaches to fusion has severe limitations. For example, Gill et al. [7] cannot fuse left folds, such as that which arises in `sum`, or `zipWith`, and Svenningsson [28] cannot handle nested computations such as mapping a function over concatenated lists. Our work is based on the stream fusion framework described by Coutts et al. [6], which can fuse all of these use cases and more. The `vector` library uses stream fusion to fuse operations on vectors rather than lists, but the principles are the same.

7. Conclusion

Generalized stream fusion is a strict improvement on stream fusion; by re-casting stream fusion to operate on bundles of streams, each vector operation or class of operations can utilize a stream representation tailored to its particular pattern of computation. We describe two new stream representations, one supporting bulk memory operations, and one adapted for SIMD computation with short-vector instructions, e.g., SSE on x86. We have added support for low-level SSE instructions to GHC and incorporated generalized stream fusion into the `vector` library. Using our modified library, programmers can write compositional, high-level programs for manipulating vectors without loss of efficiency. Benchmarks show that these programs can perform competitively with hand-written C.

Although we implemented generalized stream fusion in a Haskell library, the bundled stream representation could be used as an intermediate language in another compiler. Vector operations would no longer be first class in such a formulation, but it would allow a language to take advantage of fusion without requiring implementations of the general purpose optimizations present in GHC that allow it to eliminate the intermediate structures produced by generalized stream fusion.

Acknowledgments

The authors would like to thank Simon Marlow for his help debugging GHC's runtime system and code generator. We are also grateful for Andrew Fitzgibbon's many insightful comments.

References

[1] G. E. Blelloch, J. C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, 1994.

[2] Bryan O'Sullivan. statistics: A library of statistical types, data, and functions, aug 2012.

[3] M. M. T. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '05, page 1–13, New York, NY, USA, 2005.

[4] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: a status report. In *Proceedings of the 2007 workshop on Declarative Aspects of Multicore Programming*, DAMP '07, page 10–18, Nice, France, 2007. ACM ID: 1248652.

[5] Christian Höner zu Siederdissen. StatisticalMethods: collection of useful statistical methods., aug 2011.

[6] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: from lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 315–326, Freiburg, Germany, 2007.

[7] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional Pro-*

gramming Languages and Computer Architecture, FPCA '93, page 223–232, New York, NY, USA, 1993.

[8] K. Goto and R. v. d. Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):1–25, 2008.

[9] K. Goto and R. v. d. Geijn. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Softw.*, 35(1):1–14, 2008.

[10] G. Guennebaud, B. Jacob, and others. Eigen v3. http://eigen.tuxfamily.org, 2010.

[11] G. W. Hamilton. Extending higher-order deforestation: transforming programs to eliminate even more trees. In K. Hammond and S. Curtis, editors, *Proceedings of the Third Scottish Functional Programming Workshop*, page 25–36, Exeter, UK, UK, aug 2001.

[12] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. 2 edition, 2009.

[13] Joerg Walter, Mathias Koch, Gunter Winkler, and David Bellot. Boost basic linear algebra - 1.53.0. http://www.boost.org/doc/libs/1_53_0/libs/numeric/ublas/doc/index.htm, 2010.

[14] P. Johann. Short cut fusion: proved and improved. In *Proceedings of the 2nd international conference on Semantics, applications, and implementation of program generation*, SAIG'01, page 47–71, Berlin, Heidelberg, 2001.

[15] W. Kahan. Pracniques: further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40–, jan 1965.

[16] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, page 261–272, New York, NY, USA, 2010.

[17] J. Launchbury and S. L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.

[18] R. Leshchinskiy. vector: Efficient arrays, oct 2012.

[19] B. Lippmeier and G. Keller. Efficient parallel stencil convolution in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, page 59–70, New York, NY, USA, 2011.

[20] B. Lippmeier, M. Chakravarty, G. Keller, and S. Peyton Jones. Guiding parallel array fusion with indexed types. In *Proceedings of the 2012 Symposium on Haskell*, Haskell '12, page 25–36, New York, NY, USA, 2012.

[21] S. Marlow and S. Peyton Jones. Making a fast curry: Push/Enter vs. Eval/Apply for higher-order languages. *Journal of Functional Programming*, 16(4-5):415–449, 2006.

[22] S. Marlow and P. Wadler. Deforestation for higher-order functions. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, page 154–165, London, UK, UK, 1993.

[23] S. Peyton Jones. Call-pattern specialisation for Haskell programs. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, page 327–337, New York, NY, USA, 2007.

[24] S. Peyton Jones, R. Leshchinskiy, G. Keller, and M. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In *Programming Languages and Systems*, page 138. 2008.

[25] S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 636–666, 1991.

[26] S. L. Peyton Jones, N. Ramsey, and F. Reig. C--: a portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, sep 1999.

[27] S. L. Peyton Jones, T. Hoare, and A. Tolmach. Playing by the rules: rewriting as a practical optimisation technique. In *Proceedings of the 2001 ACM SIGPLAN Workshop on Haskell*, 2001.

[28] J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, page 124–132, New York, NY, USA, 2002.

[29] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proceedings of the seventh international conference on Functional Programming and Computer Architecture*, FPCA '95, page 306–313, New York, NY, USA, 1995.

[30] D. A. Terei and M. M. Chakravarty. An LLVM backend for GHC. In *Proceedings of the third ACM Symposium on Haskell*, Haskell '10, page 109–120, New York, NY, USA, 2010.

[31] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, jun 1995.

[32] T. L. Veldhuizen. Arrays in Blitz++. pages 223–230, dec 1998.

[33] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, jun 1990.