High Performance Linear Algebra Via Equational Transformations

Implementing BLAS libraries in Haskell

Darrell Lewis-Sandy
darrell@lewissandy.com
www.github.com/dlewissandy

March 3, 2017



©Darrell Lewis-Sandy 2017

All rights reserved. No part of this publication may be reproduced stored in a retrieval system, or transmitted in any form, or by any means, electronic, mechanical, photocopying, recording or otherwise without either the prior permission in writing from the author, or a limited use license.

The λ-blas library freely available on the world wide web and is licensed under the 3-Clause BSD license ©Darrell Lewis-Sandy 2016-2017
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The reference BLAS library is a freely-available software package. It is available from netlib via anonymous ftp and the World Wide Web. Thus, it can be included in commercial software packages (and has been). We only ask that proper credit be given to the authors.

Like all software, it is copyrighted. It is not trademarked, but we do ask the following:

- If you modify the source for these routines we ask that you change the name of the routine and comment the changes made to the original.
- We will gladly answer any questions regarding the software. If a modification is done, however, it is the responsibility of the person who modified the routine to provide support.



Section 1

Overview

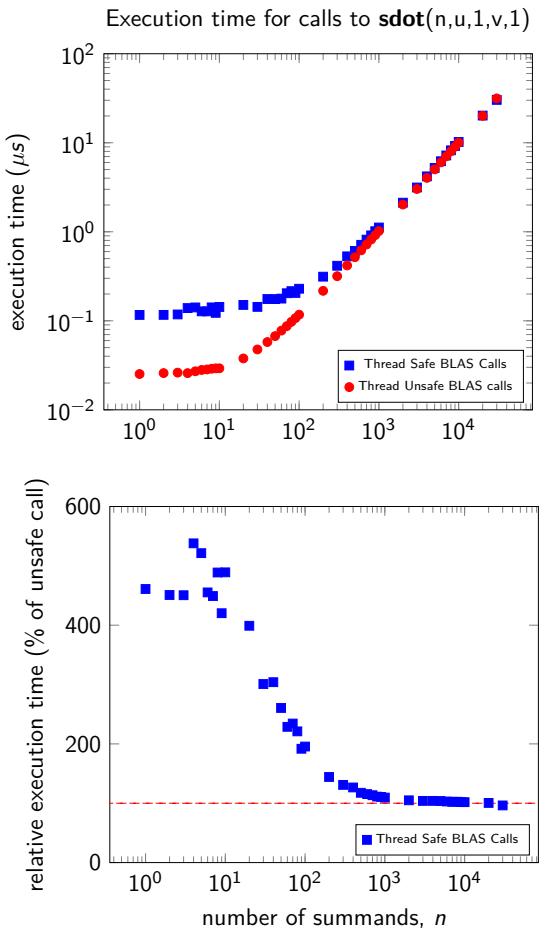
"LISP Programmers know the value of everything and the cost of nothing." *(Perlis 1982)*

Abstract

Folds, unfolds and maps are standard higher order functions for processing algebraic data structures, and provide a general framework for rapidly constructing recursive algorithms in functional programming languages. Their rigorous category-theoretic background also forms the basis for equational transformation schemes, most notably the elimination technique commonly known as "fusion". We provide an experience report on the real world usage of these concepts toward the optimization of the λ-BLAS linear algebra library. It has been our experience that it is possible to construct highly performant code without sacrificing conceptual clarity.



Motivating Example : When Using Foreign Numerical Libraries, I Need to Choose Between Speed and Thread Safety



Example: Calling Fortran BLAS 3.7.0 *sdot* function from Haskell

- Unsafe FFI calls are fast but will cause callbacks to crash and may interfere with GHC's runtime thread preemption.
- Safe calls can be used in callbacks, but may spark new OS threads and require taking locks.
- Each safe call incurs 100nS of overhead.

Overhead dominates time performance of *sdot* calls for everyday sized vectors.

The λ-BLAS Project

15097 

The λ-BLAS (lambda-BLAS) project is an effort to implement the Basic Linear Algebra Subroutines (BLAS) library in Haskell.

This work differs from previous implementations:

- Uses native Haskell code not foreign function calls.
- The functions are pure and thread safe.
- Takes advantage of universal properties of algebraic/coalgebraic datatypes and equational transformations to deforest intermediate data.

GITHUB: `www.github.com/dlewissandy/lambda-blas`
PARTICIPANTS: Darrell Lewis-Sandy, Gary Howard



Section 2

Mathematical Background

"Of the various approaches to program correctness, that of Transformational Programming appears to be the most helpful in constructing correct programs. The essence of the method is to start with an obviously correct but possibly hopelessly inefficient algorithm, and to improve it by successively applying correctness-preserving transformations" *(Meertens 1986)*

Algebraic Datatypes are a Means for Describing Structured Data

Algebraic Datatypes are built from four type-forming operations called Polynomial (Bi)functors.(Meijer, M. Fokkinga, and Paterson 1991)(Richard Bird 1997)(**Erwing1999**)

$$\begin{aligned} \textbf{Id}(\mathrm{A}) &\mapsto A \\ \textbf{Const}(\mathrm{A},\mathrm{B}) &\mapsto A \\ \textbf{Cross}(\mathrm{A},\mathrm{B}) &\mapsto A \dot{\times} B \\ \textbf{Sum}(\mathrm{A},\mathrm{B}) &\mapsto A \dot{+} B \end{aligned}$$

These type forming operations can be applied to any data types or composed to define new types and new Polynomial (Bi)functors.

(**Wadler–1990″**)(**Malcom1990**)

$$\begin{aligned} \textbf{NatF}(\mathrm{A}) &\mapsto \textbf{Const}(1,_) \dot{+} A \\ \textbf{ListF}(\mathrm{A},\mathrm{B}) &\mapsto \textbf{Const}(1,_) \dot{+} A \dot{\times} B \\ \textbf{TreeF}(\mathrm{A},\mathrm{B}) &\mapsto \textbf{Const}(1,_) \dot{+} B \dot{\times} A \dot{\times} B \end{aligned}$$



Example: Declaring Algebraic Datatypes

HASKELL

```
data Prod a b = PROD a b

data CoProd a b
    = J1 a
    | J2 b

data NatF a
    = ZERO
    | SUCC a

data ListF a b
    = NIL
    | CONS a b

data TreeF a b
    = LEAF
    | BRANCH b a b
```

SCALA

```
class Prod[A,B](var p1:A, var p2:B);

sealed trait CoProd[A,B];
case class J1[A,B]( a:A ) extends CoProd[A,B];
case class J2[A,B]( b:B ) extends CoProd[A,B];

sealed trait NatF[A];
case class ZERO[A]() extends NatF[A];
case class SUCC[A]( a:A ) extends NatF[A];

sealed trait ListF[A,B];
case class NIL[A,B]() extends ListF[A,B];
case class CONS[A,B]( a:A, b:B ) extends ListF[A,B];

sealed trait TreeF[A,B];
case class LEAF[A,B]( ) extends TreeF[A,B];
case class BRANCH[A,B]( b1:B, a::A b2:B ) extends
     TreeF[A,B];
```



The Action of Polynomial Bifunctors on Functions Provide The Basis for Equational Transformation

Polynomial (Bi)functors also act upon functions. (Maarten M Fokkinga 1994)(Richard Bird 1997)

$$
\begin{array}{lcl}
\mathbf{ID}(f) & \mapsto & f \\
\mathbf{Const}(f,g) & \mapsto & id \\
\mathbf{Cross}(f,g) & \mapsto & \lambda(a,b).(f\ a, f\ b) \\
\mathbf{Sum}(f,g) & \mapsto & \lambda(j_1\ a)\ .\ j_1\ (f\ a) \\
 & & \lambda(j_2\ b)\ .\ j_2\ (g\ b)
\end{array}
$$

Every Bifunctor,**F**, shares properties that can be used for program transformation.

$$
\begin{array}{rcll}
\mathbf{F}(id,id) & = & id & (\text{bifunctor/identity}) \\
\mathbf{F}(f,g) \circ \mathbf{F}(h,k) & = & \mathbf{F}(f \circ h, g \circ k) & (\text{bifunctor/composition})
\end{array}
$$

Example: Declaring Bifunctor Action on Functions

HASKELL

```
class Bifunctor m where
    bimap :: (a->c)->(b->d)
        ->t a b -> t c d

instance Bifunctor Prod where
    bimap f g z = case z of
        PROD a b -> PROD (f a) (g b)


instance Bifunctor CoProd where
    bimap f g z = case z of
        J1 a -> J1 (f a)
        J2 b -> J2 (g b)


instance Bifunctor ListF where
    bimap f g z = case z of
        NIL      -> NIL
        CONS a b -> CONS (f a) (g b)
```

SCALA

```
trait Bifunctor[F[_,_]] {
   def bimap[A,B,C,D]:(A=>C)=>(B=>D)
       =>F[A,B]=>F[C,D]  }

implicit object biProdF extends
     Bifunctor[Prod] {
    def bimap[A,B,C,D] = f => g => {
        case Prod(a,b) => Prod(f(a),g
            (b)) }}

implicit object biCoProdF extends
     Bifunctor[CoProd] {
    def bimap[A,B,C,D] = f => g => {
        case J1(a) => J1(f(a))
        case J2(b) => J2(g(b)) }}

implicit object biListF extends
     Bifunctor[ListF] {
    def bimap[A,B,C,D] = f => g => {
        case NIL() => NIL()
        case CONS(x,xs) => CONS(f(x),
            g(xs)) }}
```



Haskell's Rewrite Rules System can Apply Equational Transformations during Compilation

Rules are expressed as compiler pragmas(S. Peyton Jones, Tolmach, and Hoare 2001):

```
{-# RULES
"bimap/id" bimap id id = id
"bimap/bimap" forall f g h k z .
    bimap f g (bimap h k z) = bimap (f.h) (g.k) z
    #-}
```

Different libraries support different rewrite rules for algebraic types

- Stream Fusion (Coutts, Leshchinskiy, and Stewart 2007)
- Generalized Stream Fusion (Mainland, Leshchinskiy, and Peyton Jones 2013)
- Shortcut Fusion (Gill, Launchbury, and Peyton Jones 1993)
- Datatype Generics (Gibbons 2007)

They are all based on the calculus of polynomial bifunctors.



Example - Application of Compiler Rules Pragmas In Haskell

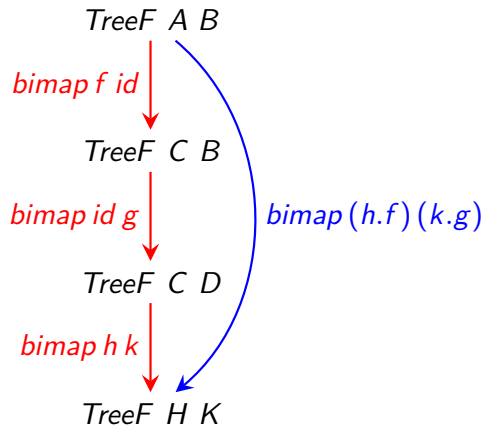
Haskell Source Code

```
foo :: TreeF a b -> TreeF h k
foo = bimap h k . bimap id g . bimap f id
```

Reduction Sequence

$$
\begin{aligned}
foo = \quad & bimap\ h\ k . bimap\ id\ g . bimap\ f\ id \\
\wr \quad & bimap/composition \\
= \quad & bimap\ h\ k . bimap\ (id . f)\ (g . id) \\
\wr \quad & identity \\
= \quad & bimap\ h\ k . bimap\ f\ g \\
\wr \quad & bimap/composition \\
= \quad & bimap\ (h . f)\ (k . g)
\end{aligned}
$$

Intermediate Tree Structures are Completely Deforested



The foo function may be subject to additional fusion with its arguments, or with downstream calling functions.

Fixed Points of Polynomial Functors are Recursive Datatypes

Every Polynomial functor* has a least fixed point $(LFix\ F_A, in)$ and a greatest fixed point $(GFix\ F_A, out)$ such that the following diagrams commute for any B, f and g.(**Manes1986**)

$$\begin{array}{ccc} F(A, LFix_{F_A}) & \xrightarrow{in_{F_A}} & LFix_{F_A} \\ {\scriptstyle F(id_A,\ fold_{F_A}\ f)}\downarrow & & \downarrow{\scriptstyle fold_{F_A}\ f} \\ F(A, B) & \xrightarrow[f]{} & B \end{array} \qquad \begin{array}{ccc} B & \xrightarrow{g} & F(A, B) \\ {\scriptstyle unfold_{F_A}\ g}\downarrow & & \downarrow{\scriptstyle F(id_A,\ unfold_{F_A}\ g)} \\ GFix_{F_A} & \xrightarrow[out_{F_A}]{} & F(A, GFix_{F_A}) \end{array}$$

The types $LFix\ F_A$ and $GFix\ F_A$ coincide* (Maarten M. Fokkinga and Meijer 1991) and in and out are bijective.

Least fixed points of polynomial functors are possibly infinite recursive datatypes* (e.g. Trees, Lists, Natural Numbers,...). (**Malcom1990**)

* in the category **FUN** defined in (Richard Bird 1997)



Examples of Recursive Datatype Declarations

Recursive Datatypes can be Declared Many Different Ways

Recursion Operator. (**Meijer1995**)

```
data Rec f = In { out :: f (Rec f) }
type List a = Rec (ListF a)
type Tree a = Rec (TreeF a)
```

Unrolled Recursive Definition.(Bird 1988)

```
data List' a = Nil | Cons a (List' a)
data Tree' a = Leaf | Branch (Tree' a
    ) a (Tree' a)
```

Church Encoding (Wadler 1990)(Gill, Launchbury, and Peyton Jones 1993)

```
data Church f = CHURCH (forall x.(f x
    ->x)->x)
type ListC a = Church (ListF a)
type TreeC a = Church (TreeF a)
```

co-Church Encoding (Wadler 1990)(Gibbons 2008)

```
data Cochurch f = forall x . COCHURCH
     (x->f x) x
type CoListC a = Cochurch (ListF a)
type CoTreeC a = Cochurch (ListF a)
```

All Declarations are Equally Expressive and Equivalent up to Isomorphism



Fold, Unfold and Map are the Fundamental Design Patterns For Recursive Programming (Gibbons2006)

fold replaces the use of a constructor with a different algebra in a way that preserves the structure.

unfold replaces the use of a destructor with a different coalgebra in a way that preserves structure.

map applies a function over the elements of a recursive type in a way that preserves the structure.

```
fold :: (Bifunctor f) => (f a b -> b)
    -> Rec (f a) -> b
fold g = g . bimap id (fold g) . out
```

```
unfold :: (Bifunctor f) => (b -> f a b)
    -> b -> Rec (f a)
unfold g = In . bimap id (unfold g) . g
```

```
map :: (Bifunctor f) => (a -> b)
    -> Rec (f a) -> Rec (f b)
map g = In . bimap g (map g) . out
```

Lambek's Lemma provides additional equational transformations

$$in \circ out = id \quad \text{(in/out)}$$
$$out \circ in = id \quad \text{(out/in)}$$



Example: QuickSort

Haskell Source Code

```
-- sort a list
qsort :: (Ord a) => List a -> List
     a
qsort = fold join . unfold split

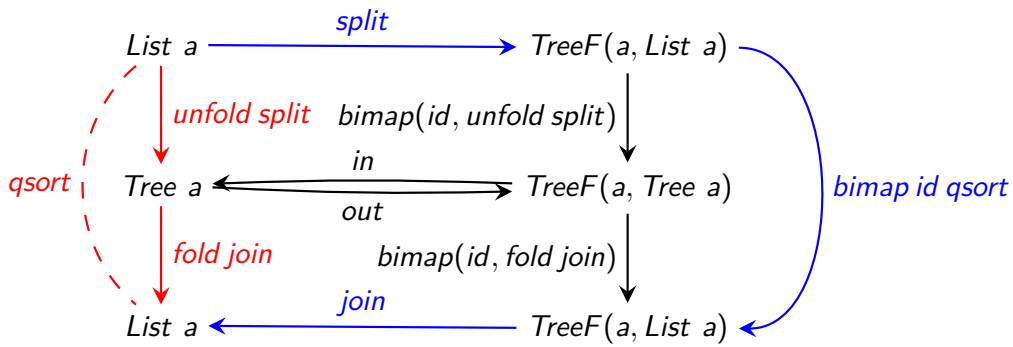
-- Partition a list on a pivot.
split :: (Ord a) => List a -> TreeF
      a (List a)
split xs = case xs of
    IN NIL -> LEAF
    IN (CONS x xs) ->
        let (ls,ges) = partition (<
             x) xs
        in  BRANCH ls x ges

-- Concatentate left list, element
-- and right list.
join :: TreeF a (List a) -> List a
join t = case t of
   LEAF -> IN NIL
   BRANCH l x r -> l `append` (IN (
        CONS x r))
```

Reduction Sequence

$$
\begin{aligned}
qsort &= fold\ join.unfold\ split \\
&\wr\ definition \\
&= join.bimap\ id\ (fold\ join).out.IN.bimap\ id\ (unfold\ split).split \\
&\wr\ isomorphism \\
&= join.bimap\ id\ (fold\ join).bimap\ id\ (unfold\ split).split \\
&\wr\ bifunctor \\
&= join.bimap\ id\ (fold\ join.unfold\ split).split \\
&\wr\ definition \\
&= join.bimap\ id\ qsort.split
\end{aligned}
$$

Intermediate Tree is Completely Deforested



qsort can be expressed as a composite of a **fold** and **unfold** through a Tree, and fused to remove the intermediate data.

Section 3

Methods

"Correctness is clearly the prime quality. If a system does not do what it is supposed to do, then everything else about it matters little." **(Bertrand Meyer)**

λ-BLAS Project Utilizes Best Practices in Test Driven Software Development

- Automated unit tests ensure byte equivalence with FORTRAN-BLAS
- Automated benchmarking using Criterion documents time performance
- Centralized source code control via GitHub
- Manditory code review via GitHub pull requests
- Continuous integration of managed code branches using TravisCI
- Agile Scrum based methodology with 1 week sprints
- Automated notification of build failures via Slack



Section 4

Results

"Talk is cheap. Show me the code" *Linus Torvalds*

Specification for the **sdot** function

Description

sdot computes the sum of n products drawn from two vectors.

The inc_u and inc_v control the increment between and the direction in which elements drawn from the two vectors.

sdot corresponds to the vector dot product when inc_u and inc_v are both equal to one.

Mathematical Specification

$\forall n \in \mathbb{N}$
$\forall inc_u, inc_v \in \mathbb{Z}$
$\forall m_u, m_v \in \{max(inc_u, inc_v)(n-1)\dots\}$
$\forall u \in \mathbb{R}^{m_u}$
$\forall v \in \mathbb{R}^{m_v}$

$$\textbf{sdot}\ n\ v\ inc_u\ v\ inc_v = \sum_{i=0}^{n-1} u_{[first(inc_u)+i\ \ inc_u]} v_{[first(inc_v)+i\ \ inc_v]}$$

$$\textbf{where}\quad first\ inc = \begin{cases} 0 & inc > 0 \\ (1-n)inc & inc \leq 0 \end{cases}$$

Specification for the **sdot** function

Description

sdot computes the sum of n products drawn from two vectors.

The inc_u and inc_v control the increment between and the direction in which elements drawn from the two vectors.

sdot corresponds to the vector dot product when inc_u and inc_v are both equal to one.

Implementation Details

- Written in FORTRAN-90

Fortran Source Code

```
      REAL FUNCTION SDOT(N,SX,INCX,SY,INCY)
*  -- Reference BLAS level1 routine (version 3.7.0) --
*  -- Reference BLAS is a software package provided by Univ. of Tennessee,    --
*  -- Univ. of California Berkeley, Univ. of Colorado Denver and NAG Ltd..--
*     December 2016
      INTEGER INCX,INCY,N
      REAL SX(*),SY(*)
      REAL STEMP
      INTEGER I,IX,IY,M,MP1
      INTRINSIC MOD
      STEMP = 0.0e0
      SDOT = 0.0e0
      IF (N.LE.0) RETURN
      IF (INCX.EQ.1 .AND. INCY.EQ.1) THEN
*        code for both increments equal to 1
         M = MOD(N,5)
         IF (M.NE.0) THEN
            DO I = 1,M
               STEMP = STEMP + SX(I)*SY(I)
            END DO
            IF (N.LT.5) THEN
               SDOT=STEMP
            RETURN
            END IF
         END IF
         MP1 = M + 1
         DO I = MP1,N,5
          STEMP = STEMP + SX(I)*SY(I) + SX(I+1)*SY(I+1) +
     $            SX(I+2)*SY(I+2) + SX(I+3)*SY(I+3) + SX(I+4)*SY(I+4)
         END DO
      ELSE
*        code for unequal increments or equal increments
*          not equal to 1
         IX = 1
         IY = 1
         IF (INCX.LT.0) IX = (-N+1)*INCX + 1
         IF (INCY.LT.0) IY = (-N+1)*INCY + 1
         DO I = 1,N
            STEMP = STEMP + SX(IX)*SY(IY)
            IX = IX + INCX
            IY = IY + INCY
         END DO
      END IF
      SDOT = STEMP
      RETURN
      END
```



Specification for the **sdot** function

Description

sdot computes the sum of n products drawn from two vectors.

The inc_u and inc_v control the increment between and the direction in which elements drawn from the two vectors.

sdot corresponds to the vector dot product when inc_u and inc_v are both equal to one.

Implementation Details

- Written in FORTRAN-90
- Vectors are represented as unboxed one indexed arrays

Fortran Source Code

```
      REAL FUNCTION SDOT(N,SX,INCX,SY,INCY)
*  -- Reference BLAS level1 routine (version 3.7.0) --
*  -- Reference BLAS is a software package provided by Univ. of Tennessee,    --
*  -- Univ. of California Berkeley, Univ. of Colorado Denver and NAG Ltd..--
*     December 2016
      INTEGER INCX,INCY,N
      REAL SX(*),SY(*)
      REAL STEMP
      INTEGER I,IX,IY,M,MP1
      INTRINSIC MOD
      STEMP = 0.0e0
      SDOT = 0.0e0
      IF (N.LE.0) RETURN
      IF (INCX.EQ.1 .AND. INCY.EQ.1) THEN
*        code for both increments equal to 1
         M = MOD(N,5)
         IF (M.NE.0) THEN
            DO I = 1,M
               STEMP = STEMP + SX(I)*SY(I)
            END DO
            IF (N.LT.5) THEN
               SDOT=STEMP
            RETURN
            END IF
         END IF
         MP1 = M + 1
         DO I = MP1,N,5
          STEMP = STEMP + SX(I)*SY(I) + SX(I+1)*SY(I+1) +
     $            SX(I+2)*SY(I+2) + SX(I+3)*SY(I+3) + SX(I+4)*SY(I+4)
         END DO
      ELSE
*        code for unequal increments or equal increments
*          not equal to 1
         IX = 1
         IY = 1
         IF (INCX.LT.0) IX = (-N+1)*INCX + 1
         IF (INCY.LT.0) IY = (-N+1)*INCY + 1
         DO I = 1,N
            STEMP = STEMP + SX(IX)*SY(IY)
            IX = IX + INCX
            IY = IY + INCY
         END DO
      END IF
      SDOT = STEMP
      RETURN
      END
```



Specification for the **sdot** function

Description

sdot computes the sum of n products drawn from two vectors.

The inc_u and inc_v control the increment between and the direction in which elements drawn from the two vectors.

sdot corresponds to the vector dot product when inc_u and inc_v are both equal to one.

Implementation Details

- Written in FORTRAN-90
- Vectors are represented as unboxed one indexed arrays
- O(1) Special case calculation when no summands are required.

Fortran Source Code

```
      REAL FUNCTION SDOT(N,SX,INCX,SY,INCY)
*  -- Reference BLAS level1 routine (version 3.7.0) --
*  -- Reference BLAS is a software package provided by Univ. of Tennessee,    --
*  -- Univ. of California Berkeley, Univ. of Colorado Denver and NAG Ltd..--
*     December 2016
      INTEGER INCX,INCY,N
      REAL SX(*),SY(*)
      REAL STEMP
      INTEGER I,IX,IY,M,MP1
      INTRINSIC MOD
      STEMP = 0.0e0
      SDOT = 0.0e0
      IF (N.LE.0) RETURN
      IF (INCX.EQ.1 .AND. INCY.EQ.1) THEN
*        code for both increments equal to 1
         M = MOD(N,5)
         IF (M.NE.0) THEN
            DO I = 1,M
               STEMP = STEMP + SX(I)*SY(I)
            END DO
            IF (N.LT.5) THEN
               SDOT=STEMP
            RETURN
            END IF
         END IF
         MP1 = M + 1
         DO I = MP1,N,5
          STEMP = STEMP + SX(I)*SY(I) + SX(I+1)*SY(I+1) +
     $            SX(I+2)*SY(I+2) + SX(I+3)*SY(I+3) + SX(I+4)*SY(I+4)
         END DO
      ELSE
*        code for unequal increments or equal increments
*          not equal to 1
         IX = 1
         IY = 1
         IF (INCX.LT.0) IX = (-N+1)*INCX + 1
         IF (INCY.LT.0) IY = (-N+1)*INCY + 1
         DO I = 1,N
            STEMP = STEMP + SX(IX)*SY(IY)
            IX = IX + INCX
            IY = IY + INCY
         END DO
      END IF
      SDOT = STEMP
      RETURN
      END
```



Specification for the **sdot** function

Description

sdot computes the sum of n products drawn from two vectors.

The inc_u and inc_v control the increment between and the direction in which elements drawn from the two vectors.

sdot corresponds to the vector dot product when inc_u and inc_v are both equal to one.

Implementation Details

- Written in FORTRAN-90
- Vectors are represented as unboxed one indexed arrays
- O(1) Special case calculation when no summands are required.
- O(n) Special case calculation when both inc_u and inc_v are one.

Fortran Source Code

```
      REAL FUNCTION SDOT(N,SX,INCX,SY,INCY)
*  -- Reference BLAS level1 routine (version 3.7.0) --
*  -- Reference BLAS is a software package provided by Univ. of Tennessee,    --
*  -- Univ. of California Berkeley, Univ. of Colorado Denver and NAG Ltd..--
*     December 2016
      INTEGER INCX,INCY,N
      REAL SX(*),SY(*)
      REAL STEMP
      INTEGER I,IX,IY,M,MP1
      INTRINSIC MOD
      STEMP = 0.0e0
      SDOT = 0.0e0
      IF (N.LE.0) RETURN
      IF (INCX.EQ.1 .AND. INCY.EQ.1) THEN
*        code for both increments equal to 1
         M = MOD(N,5)
         IF (M.NE.0) THEN
            DO I = 1,M
               STEMP = STEMP + SX(I)*SY(I)
            END DO
            IF (N.LT.5) THEN
               SDOT=STEMP
            RETURN
            END IF
         END IF
         MP1 = M + 1
         DO I = MP1,N,5
          STEMP = STEMP + SX(I)*SY(I) + SX(I+1)*SY(I+1) +
     $            SX(I+2)*SY(I+2) + SX(I+3)*SY(I+3) + SX(I+4)*SY(I+4)
         END DO
      ELSE
*        code for unequal increments or equal increments
*          not equal to 1
         IX = 1
         IY = 1
         IF (INCX.LT.0) IX = (-N+1)*INCX + 1
         IF (INCY.LT.0) IY = (-N+1)*INCY + 1
         DO I = 1,N
            STEMP = STEMP + SX(IX)*SY(IY)
            IX = IX + INCX
            IY = IY + INCY
         END DO
      END IF
      SDOT = STEMP
      RETURN
      END
```

Specification for the **sdot** function

Description

sdot computes the sum of n products drawn from two vectors.

The inc_u and inc_v control the increment between and the direction in which elements drawn from the two vectors.

sdot corresponds to the vector dot product when inc_u and inc_v are both equal to one.

Implementation Details

- Written in FORTRAN-90
- Vectors are represented as unboxed one indexed arrays
- O(1) Special case calculation when no summands are required.
- O(n) Special case calculation when both inc_u and inc_v are one.
- Loop unrolling used to reduce O(n) constant when inc_u and inc_v are one.

Fortran Source Code

```
      REAL FUNCTION SDOT(N,SX,INCX,SY,INCY)
*  -- Reference BLAS level1 routine (version 3.7.0) --
*  -- Reference BLAS is a software package provided by Univ. of Tennessee,    --
*  -- Univ. of California Berkeley, Univ. of Colorado Denver and NAG Ltd..--
*     December 2016
      INTEGER INCX,INCY,N
      REAL SX(*),SY(*)
      REAL STEMP
      INTEGER I,IX,IY,M,MP1
      INTRINSIC MOD
      STEMP = 0.0e0
      SDOT = 0.0e0
      IF (N.LE.0) RETURN
      IF (INCX.EQ.1 .AND. INCY.EQ.1) THEN
*        code for both increments equal to 1
         M = MOD(N,5)
         IF (M.NE.0) THEN
            DO I = 1,M
               STEMP = STEMP + SX(I)*SY(I)
            END DO
            IF (N.LT.5) THEN
               SDOT=STEMP
            RETURN
            END IF
         END IF
         MP1 = M + 1
         DO I = MP1,N,5
          STEMP = STEMP + SX(I)*SY(I) + SX(I+1)*SY(I+1) +
     $            SX(I+2)*SY(I+2) + SX(I+3)*SY(I+3) + SX(I+4)*SY(I+4)
         END DO
      ELSE
*        code for unequal increments or equal increments
*          not equal to 1
         IX = 1
         IY = 1
         IF (INCX.LT.0) IX = (-N+1)*INCX + 1
         IF (INCY.LT.0) IY = (-N+1)*INCY + 1
         DO I = 1,N
            STEMP = STEMP + SX(IX)*SY(IY)
            IX = IX + INCX
            IY = IY + INCY
         END DO
      END IF
      SDOT = STEMP
      RETURN
      END
```

Specification for the **sdot** function

Description

sdot computes the sum of n products drawn from two vectors.

The inc_u and inc_v control the increment between and the direction in which elements drawn from the two vectors.

sdot corresponds to the vector dot product when inc_u and inc_v are both equal to one.

Implementation Details

- Written in FORTRAN-90
- Vectors are represented as unboxed one indexed arrays
- O(1) Special case calculation when no summands are required.
- O(n) Special case calculation when both inc_u and inc_v are one.
- Loop unrolling used to reduce O(n) constant when inc_u and inc_v are one.
- O(n) Default case calculation for all other cases.

Fortran Source Code

```
      REAL FUNCTION SDOT(N,SX,INCX,SY,INCY)
*  -- Reference BLAS level1 routine (version 3.7.0) --
*  -- Reference BLAS is a software package provided by Univ. of Tennessee,    --
*  -- Univ. of California Berkeley, Univ. of Colorado Denver and NAG Ltd..--
*     December 2016
      INTEGER INCX,INCY,N
      REAL SX(*),SY(*)
      REAL STEMP
      INTEGER I,IX,IY,M,MP1
      INTRINSIC MOD
      STEMP = 0.0e0
      SDOT = 0.0e0
      IF (N.LE.0) RETURN
      IF (INCX.EQ.1 .AND. INCY.EQ.1) THEN
*        code for both increments equal to 1
         M = MOD(N,5)
         IF (M.NE.0) THEN
            DO I = 1,M
               STEMP = STEMP + SX(I)*SY(I)
            END DO
            IF (N.LT.5) THEN
               SDOT=STEMP
            RETURN
            END IF
         END IF
         MP1 = M + 1
         DO I = MP1,N,5
          STEMP = STEMP + SX(I)*SY(I) + SX(I+1)*SY(I+1) +
     $            SX(I+2)*SY(I+2) + SX(I+3)*SY(I+3) + SX(I+4)*SY(I+4)
         END DO
      ELSE
*        code for unequal increments or equal increments
*          not equal to 1
         IX = 1
         IY = 1
         IF (INCX.LT.0) IX = (-N+1)*INCX + 1
         IF (INCY.LT.0) IY = (-N+1)*INCY + 1
         DO I = 1,N
            STEMP = STEMP + SX(IX)*SY(IY)
            IX = IX + INCX
            IY = IY + INCY
         END DO
      END IF
      SDOT = STEMP
      RETURN
      END
```



Specification for the **sdot** function

Description

sdot computes the sum of n products drawn from two vectors.

The inc_u and inc_v control the increment between and the direction in which elements drawn from the two vectors.

sdot corresponds to the vector dot product when inc_u and inc_v are both equal to one.

Implementation Details

- Written in FORTRAN-90
- Vectors are represented as unboxed one indexed arrays
- O(1) Special case calculation when no summands are required.
- O(n) Special case calculation when both inc_u and inc_v are one.
- Loop unrolling used to reduce O(n) constant when inc_u and inc_v are one.
- O(n) Default case calculation for all other cases.
- Calling program assumes responsibility for ensuring vectors are long enough.

Fortran Source Code

```
      REAL FUNCTION SDOT(N,SX,INCX,SY,INCY)
*  -- Reference BLAS level1 routine (version 3.7.0) --
*  -- Reference BLAS is a software package provided by Univ. of Tennessee,    --
*  -- Univ. of California Berkeley, Univ. of Colorado Denver and NAG Ltd..--
*     December 2016
      INTEGER INCX,INCY,N
      REAL SX(*),SY(*)
      REAL STEMP
      INTEGER I,IX,IY,M,MP1
      INTRINSIC MOD
      STEMP = 0.0e0
      SDOT = 0.0e0
      IF (N.LE.0) RETURN
      IF (INCX.EQ.1 .AND. INCY.EQ.1) THEN
*        code for both increments equal to 1
         M = MOD(N,5)
         IF (M.NE.0) THEN
            DO I = 1,M
               STEMP = STEMP + SX(I)*SY(I)
            END DO
            IF (N.LT.5) THEN
               SDOT=STEMP
            RETURN
            END IF
         END IF
         MP1 = M + 1
         DO I = MP1,N,5
          STEMP = STEMP + SX(I)*SY(I) + SX(I+1)*SY(I+1) +
     $            SX(I+2)*SY(I+2) + SX(I+3)*SY(I+3) + SX(I+4)*SY(I+4)
         END DO
      ELSE
*        code for unequal increments or equal increments
*          not equal to 1
         IX = 1
         IY = 1
         IF (INCX.LT.0) IX = (-N+1)*INCX + 1
         IF (INCY.LT.0) IY = (-N+1)*INCY + 1
         DO I = 1,N
            STEMP = STEMP + SX(IX)*SY(IY)
            IX = IX + INCX
            IY = IY + INCY
         END DO
      END IF
      SDOT = STEMP
      RETURN
      END
```



Initial Implementation : "Short Cut" Fusion on Lists

FUSION STRATEGY

List producing functions (e.g. take, zipWith, replicate), are abstracted as "builds".(Gill, Launchbury, and Peyton Jones 1993)

List consuming functions are (mostly) represented as right associative folds.

The list producers can be fused to list consumers using the foldr/build rule to deforest intermediate list structures.

Mathematical Specification

$$\forall g \in \forall \beta.(A \to \beta \to \beta) \to \beta -> \to \beta$$
$$\forall f \in A \to B \to B$$
$$\forall b \; in \; B$$
$$\texttt{build}\, g = g \; CONS \; NIL$$
$$\texttt{foldr}\, f \; b \circ build \; g = g \; f \; b$$

Initial Implementation : "Short Cut" Fusion on Lists

FUSION STRATEGY

List producing functions (e.g. take, zipWith, replicate), are abstracted as "builds".(Gill, Launchbury, and Peyton Jones 1993)

List consuming functions are (mostly) represented as right associative folds.

The list producers can be fused to list consumers using the foldr/build rule to deforest intermediate list structures.

IMPLEMENTATION STRATEGY

- Written in Haskell-2010

Haskell Source Code

```
import Numerical.BLAS.Types
import Numerical.BLAS.Util

import Data.Ord(comparing)
import Data.Vector.Storable(Vector)
import qualified Data.Vector.Storable as V

-- | O(n) - compute the sum of products of elements drawn from two Float
     vectors.
-- This function is morally equivalent to the dot product of vectors x and
     y
-- when n is the length of both x and y, and both incx and incy are unity.
dot :: (V.Storable a, Num a)
    => Int       -- ^ The number of elements n
    -> Vector a -- ^ The first vector, x
    -> Int       -- ^ The increment, incx, between elements drawn from x
    -> Vector a -- ^ The second vector, y
    -> Int       -- ^ The increment, incy,  between elements drawn from y
    -> a         -- ^ The sum of products of the selected elements
{-# INLINE dot #-}
dot !n sx !incx sy !incy = V.foldl' (+) 0
    $ V.zipWith (*) (sampleElems n sx incx) $ sampleElems n sy incy
{-# DEPRECATED sdot,ddot "Use dot instead." #-}
sdot :: Int -> Vector Float -> Int -> Vector Float -> Int -> Float
sdot = dot
ddot :: Int -> Vector Double -> Int -> Vector Double -> Int -> Double
ddot = dot

-- O(1) - Compute the starting index of an iterative vector traversal
-- given the length of the vector and the iterative step size.
firstIndex :: Int -> Int -- ^ the iterative step
           -> Int -- ^ The index of the first iterative step
{-# INLINE firstIndex #-}
firstIndex !n !inc
    | inc>0     = 0
    | otherwise = (1-n)*inc

-- | O(n) - Multiply selected elements in a vector, x, by a scalar, a.
     This
-- function is morally equivalent to scalar multiplication a*x when n is
     the
-- length x and incx is unity.
scal :: (V.Storable a, Num a)
    => Int       -- ^ The number of elements to scale, n
    -> a         -- ^ The scalar factor, a
```

Initial Implementation : "Short Cut" Fusion on Lists

FUSION STRATEGY

List producing functions (e.g. take, zipWith, replicate), are abstracted as "builds".(Gill, Launchbury, and Peyton Jones 1993)

List consuming functions are (mostly) represented as right associative folds.

The list producers can be fused to list consumers using the foldr/build rule to deforest intermediate list structures.

IMPLEMENTATION STRATEGY

- Written in Haskell-2010
- Vectors are represented as Haskell's built in singly-linked list type

Haskell Source Code

```
import Numerical.BLAS.Types
import Numerical.BLAS.Util

import Data.Ord(comparing)
import Data.Vector.Storable(Vector)
import qualified Data.Vector.Storable as V

-- | O(n) - compute the sum of products of elements drawn from two Float
     vectors.
-- This function is morally equivalent to the dot product of vectors x and
     y
-- when n is the length of both x and y, and both incx and incy are unity.
dot :: (V.Storable a, Num a)
    => Int       -- ^ The number of elements n
    -> Vector a -- ^ The first vector, x
    -> Int       -- ^ The increment, incx, between elements drawn from x
    -> Vector a -- ^ The second vector, y
    -> Int       -- ^ The increment, incy,  between elements drawn from y
    -> a         -- ^ The sum of products of the selected elements
{-# INLINE dot #-}
dot !n sx !incx sy !incy = V.foldl' (+) 0
    $ V.zipWith (*) (sampleElems n sx incx) $ sampleElems n sy incy
{-# DEPRECATED sdot,ddot "Use dot instead." #-}
sdot :: Int -> Vector Float -> Int -> Vector Float -> Int -> Float
sdot = dot
ddot :: Int -> Vector Double -> Int -> Vector Double -> Int -> Double
ddot = dot

-- O(1) - Compute the starting index of an iterative vector traversal
-- given the length of the vector and the iterative step size.
firstIndex :: Int -> Int -- ^ the iterative step
           -> Int -- ^ The index of the first iterative step
{-# INLINE firstIndex #-}
firstIndex !n !inc
    | inc>0     = 0
    | otherwise = (1-n)*inc

-- | O(n) - Multiply selected elements in a vector, x, by a scalar, a.
     This
-- function is morally equivalent to scalar multiplication a*x when n is
     the
-- length x and incx is unity.
scal :: (V.Storable a, Num a)
    => Int       -- ^ The number of elements to scale, n
    -> a         -- ^ The scalar factor, a
```

Initial Implementation : "Short Cut" Fusion on Lists

FUSION STRATEGY

List producing functions (e.g. take, zipWith, replicate), are abstracted as "builds".(Gill, Launchbury, and Peyton Jones 1993)

List consuming functions are (mostly) represented as right associative folds.

The list producers can be fused to list consumers using the foldr/build rule to deforest intermediate list structures.

IMPLEMENTATION STRATEGY

- Written in Haskell-2010
- Vectors are represented as Haskell's built in singly-linked list type
- O(n) When both inc_x and inc_y are zero, compute the solution as a fold/replicate composite.

Haskell Source Code

```
import Numerical.BLAS.Types
import Numerical.BLAS.Util

import Data.Ord(comparing)
import Data.Vector.Storable(Vector)
import qualified Data.Vector.Storable as V

-- | O(n) - compute the sum of products of elements drawn from two Float
     vectors.
-- This function is morally equivalent to the dot product of vectors x and
     y
-- when n is the length of both x and y, and both incx and incy are unity.
dot :: (V.Storable a, Num a)
    => Int       -- ^ The number of elements n
    -> Vector a -- ^ The first vector, x
    -> Int       -- ^ The increment, incx, between elements drawn from x
    -> Vector a -- ^ The second vector, y
    -> Int       -- ^ The increment, incy,  between elements drawn from y
    -> a         -- ^ The sum of products of the selected elements
{-# INLINE dot #-}
dot !n sx !incx sy !incy = V.foldl' (+) 0
    $ V.zipWith (*) (sampleElems n sx incx) $ sampleElems n sy incy
{-# DEPRECATED sdot,ddot "Use dot instead." #-}
sdot :: Int -> Vector Float -> Int -> Vector Float -> Int -> Float
sdot = dot
ddot :: Int -> Vector Double -> Int -> Vector Double -> Int -> Double
ddot = dot

-- O(1) - Compute the starting index of an iterative vector traversal
-- given the length of the vector and the iterative step size.
firstIndex :: Int -> Int -- ^ the iterative step
           -> Int -- ^ The index of the first iterative step
{-# INLINE firstIndex #-}
firstIndex !n !inc
    | inc>0     = 0
    | otherwise = (1-n)*inc

-- | O(n) - Multiply selected elements in a vector, x, by a scalar, a.
     This
-- function is morally equivalent to scalar multiplication a*x when n is
     the
-- length x and incx is unity.
scal :: (V.Storable a, Num a)
    => Int       -- ^ The number of elements to scale, n
    -> a         -- ^ The scalar factor, a
```

Initial Implementation : "Short Cut" Fusion on Lists

FUSION STRATEGY

List producing functions (e.g. take, zipWith, replicate), are abstracted as "builds".(Gill, Launchbury, and Peyton Jones 1993)

List consuming functions are (mostly) represented as right associative folds.

The list producers can be fused to list consumers using the foldr/build rule to deforest intermediate list structures.

IMPLEMENTATION STRATEGY

- Written in Haskell-2010
- Vectors are represented as Haskell's built in singly-linked list type
- O(n) When both inc_x and inc_y are zero, compute the solution as a fold/replicate composite.
- O(n) When both vectors are being traversed in the same direction and no elements are skipped, then compute solution as fold/take/zipWith composite.

Haskell Source Code

```
import Numerical.BLAS.Types
import Numerical.BLAS.Util

import Data.Ord(comparing)
import Data.Vector.Storable(Vector)
import qualified Data.Vector.Storable as V

-- | O(n) - compute the sum of products of elements drawn from two Float
     vectors.
-- This function is morally equivalent to the dot product of vectors x and
     y
-- when n is the length of both x and y, and both incx and incy are unity.
dot :: (V.Storable a, Num a)
    => Int       -- ^ The number of elements n
    -> Vector a -- ^ The first vector, x
    -> Int       -- ^ The increment, incx, between elements drawn from x
    -> Vector a -- ^ The second vector, y
    -> Int       -- ^ The increment, incy,  between elements drawn from y
    -> a         -- ^ The sum of products of the selected elements
{-# INLINE dot #-}
dot !n sx !incx sy !incy = V.foldl' (+) 0
    $ V.zipWith (*) (sampleElems n sx incx) $ sampleElems n sy incy
{-# DEPRECATED sdot,ddot "Use dot instead." #-}
sdot :: Int -> Vector Float -> Int -> Vector Float -> Int -> Float
sdot = dot
ddot :: Int -> Vector Double -> Int -> Vector Double -> Int -> Double
ddot = dot

-- O(1) - Compute the starting index of an iterative vector traversal
-- given the length of the vector and the iterative step size.
firstIndex :: Int -> Int -- ^ the iterative step
           -> Int -- ^ The index of the first iterative step
{-# INLINE firstIndex #-}
firstIndex !n !inc
    | inc>0     = 0
    | otherwise = (1-n)*inc

-- | O(n) - Multiply selected elements in a vector, x, by a scalar, a.
     This
-- function is morally equivalent to scalar multiplication a*x when n is
     the
-- length x and incx is unity.
scal :: (V.Storable a, Num a)
    => Int       -- ^ The number of elements to scale, n
    -> a         -- ^ The scalar factor, a
```

Initial Implementation : "Short Cut" Fusion on Lists

FUSION STRATEGY

List producing functions (e.g. take, zipWith, replicate), are abstracted as "builds".(Gill, Launchbury, and Peyton Jones 1993)

List consuming functions are (mostly) represented as right associative folds.

The list producers can be fused to list consumers using the foldr/build rule to deforest intermediate list structures.

IMPLEMENTATION STRATEGY

- Written in Haskell-2010
- Vectors are represented as Haskell's built in singly-linked list type
- O(n) When both inc_x and inc_y are zero, compute the solution as a fold/replicate composite.
- O(n) When both vectors are being traversed in the same direction and no elements are skipped, then compute solution as fold/take/zipWith composite.
- O(n) For all other cases, express the solution as a fold/getProds composite.

Haskell Source Code

```
import Numerical.BLAS.Types
import Numerical.BLAS.Util

import Data.Ord(comparing)
import Data.Vector.Storable(Vector)
import qualified Data.Vector.Storable as V

-- | O(n) - compute the sum of products of elements drawn from two Float
     vectors.
-- This function is morally equivalent to the dot product of vectors x and
     y
-- when n is the length of both x and y, and both incx and incy are unity.
dot :: (V.Storable a, Num a)
    => Int       -- ^ The number of elements n
    -> Vector a -- ^ The first vector, x
    -> Int       -- ^ The increment, incx, between elements drawn from x
    -> Vector a -- ^ The second vector, y
    -> Int       -- ^ The increment, incy,  between elements drawn from y
    -> a         -- ^ The sum of products of the selected elements
{-# INLINE dot #-}
dot !n sx !incx sy !incy = V.foldl' (+) 0
    $ V.zipWith (*) (sampleElems n sx incx) $ sampleElems n sy incy
{-# DEPRECATED sdot,ddot "Use dot instead." #-}
sdot :: Int -> Vector Float -> Int -> Vector Float -> Int -> Float
sdot = dot
ddot :: Int -> Vector Double -> Int -> Vector Double -> Int -> Double
ddot = dot

-- O(1) - Compute the starting index of an iterative vector traversal
-- given the length of the vector and the iterative step size.
firstIndex :: Int -> Int -- ^ the iterative step
           -> Int -- ^ The index of the first iterative step
{-# INLINE firstIndex #-}
firstIndex !n !inc
    | inc>0     = 0
    | otherwise = (1-n)*inc

-- | O(n) - Multiply selected elements in a vector, x, by a scalar, a.
     This
-- function is morally equivalent to scalar multiplication a*x when n is
     the
-- length x and incx is unity.
scal :: (V.Storable a, Num a)
    => Int       -- ^ The number of elements to scale, n
    -> a         -- ^ The scalar factor, a
```

Initial Implementation : "Short Cut" Fusion on Lists

FUSION STRATEGY

List producing functions (e.g. take, zipWith, replicate), are abstracted as "builds".(Gill, Launchbury, and Peyton Jones 1993)

List consuming functions are (mostly) represented as right associative folds.

The list producers can be fused to list consumers using the foldr/build rule to deforest intermediate list structures.

IMPLEMENTATION STRATEGY

- Written in Haskell-2010
- Vectors are represented as Haskell's built in singly-linked list type
- O(n) When both inc_x and inc_y are zero, compute the solution as a fold/replicate composite.
- O(n) When both vectors are being traversed in the same direction and no elements are skipped, then compute solution as fold/take/zipWith composite.
- O(n) For all other cases, express the solution as a fold/getProds composite.
- Vectors are reversed when the direction of traversals are not the same to avoid O(n^2) time complexity.

Haskell Source Code

```
import Numerical.BLAS.Types
import Numerical.BLAS.Util

import Data.Ord(comparing)
import Data.Vector.Storable(Vector)
import qualified Data.Vector.Storable as V

-- | O(n) - compute the sum of products of elements drawn from two Float
     vectors.
-- This function is morally equivalent to the dot product of vectors x and
     y
-- when n is the length of both x and y, and both incx and incy are unity.
dot :: (V.Storable a, Num a)
    => Int       -- ^ The number of elements n
    -> Vector a -- ^ The first vector, x
    -> Int       -- ^ The increment, incx, between elements drawn from x
    -> Vector a -- ^ The second vector, y
    -> Int       -- ^ The increment, incy,  between elements drawn from y
    -> a         -- ^ The sum of products of the selected elements
{-# INLINE dot #-}
dot !n sx !incx sy !incy = V.foldl' (+) 0
    $ V.zipWith (*) (sampleElems n sx incx) $ sampleElems n sy incy
{-# DEPRECATED sdot,ddot "Use dot instead." #-}
sdot :: Int -> Vector Float -> Int -> Vector Float -> Int -> Float
sdot = dot
ddot :: Int -> Vector Double -> Int -> Vector Double -> Int -> Double
ddot = dot

-- O(1) - Compute the starting index of an iterative vector traversal
-- given the length of the vector and the iterative step size.
firstIndex :: Int -> Int -- ^ the iterative step
           -> Int -- ^ The index of the first iterative step
{-# INLINE firstIndex #-}
firstIndex !n !inc
    | inc>0     = 0
    | otherwise = (1-n)*inc

-- | O(n) - Multiply selected elements in a vector, x, by a scalar, a.
     This
-- function is morally equivalent to scalar multiplication a*x when n is
     the
-- length x and incx is unity.
scal :: (V.Storable a, Num a)
    => Int       -- ^ The number of elements to scale, n
    -> a         -- ^ The scalar factor, a
```

Initial Implementation : "Short Cut" Fusion on Lists

FUSION STRATEGY

List producing functions (e.g. take, zipWith, replicate), are abstracted as "builds".(Gill, Launchbury, and Peyton Jones 1993)

List consuming functions are (mostly) represented as right associative folds.

The list producers can be fused to list consumers using the foldr/build rule to deforest intermediate list structures.

IMPLEMENTATION STRATEGY

- Written in Haskell-2010
- Vectors are represented as Haskell's built in singly-linked list type
- O(n) When both inc_x and inc_y are zero, compute the solution as a fold/replicate composite.
- O(n) When both vectors are being traversed in the same direction and no elements are skipped, then compute solution as fold/take/zipWith composite.
- O(n) For all other cases, express the solution as a fold/getProds composite.
- Vectors are reversed when the direction of traversals are not the same to avoid $O(n^2)$ time complexity.
- Calling program assumes responsibility for ensuring vectors are long enough.

Haskell Source Code

```
import Numerical.BLAS.Types
import Numerical.BLAS.Util

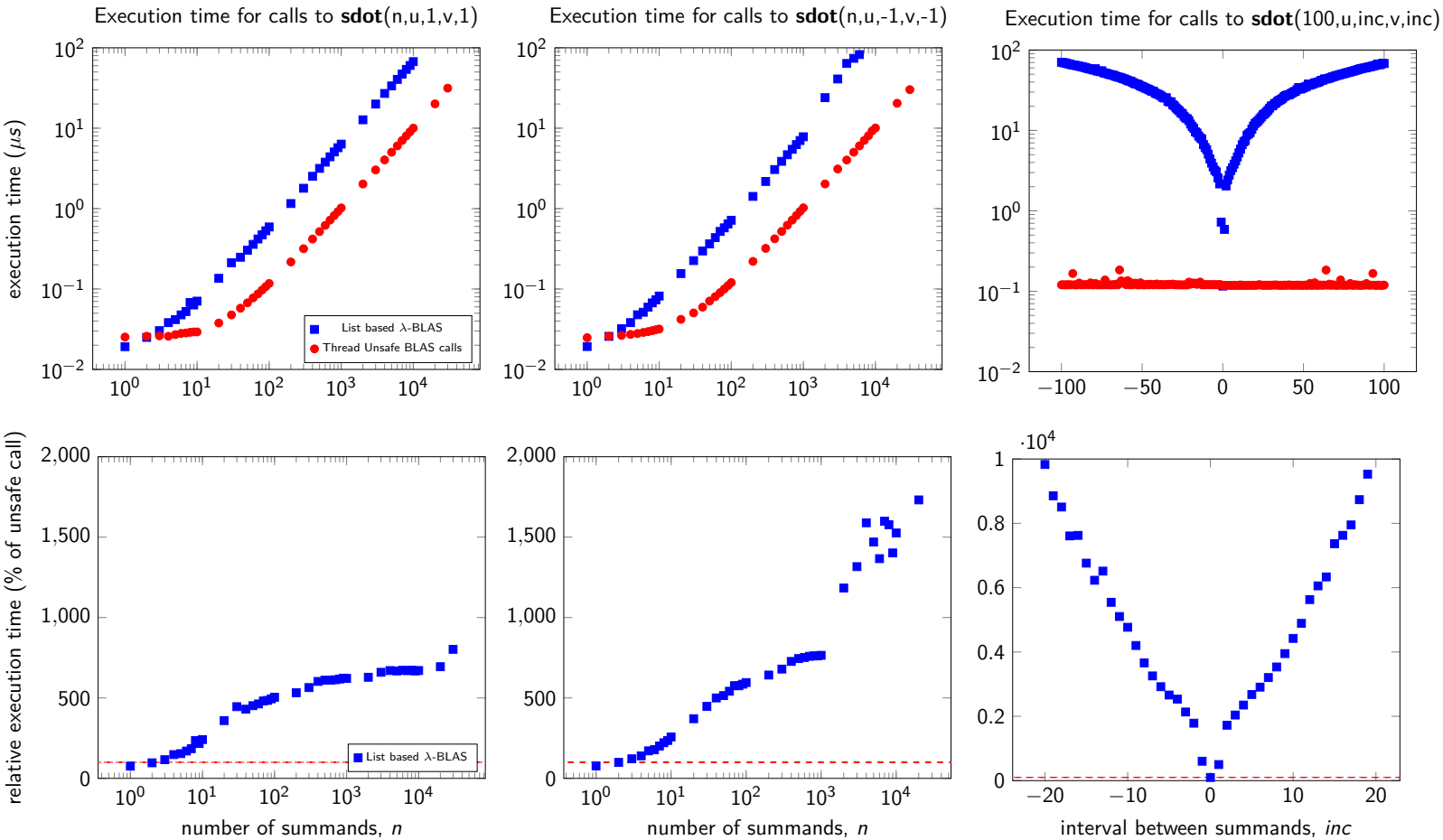
import Data.Ord(comparing)
import Data.Vector.Storable(Vector)
import qualified Data.Vector.Storable as V

-- | O(n) - compute the sum of products of elements drawn from two Float
     vectors.
-- This function is morally equivalent to the dot product of vectors x and
     y
-- when n is the length of both x and y, and both incx and incy are unity.
dot :: (V.Storable a, Num a)
    => Int       -- ^ The number of elements n
    -> Vector a -- ^ The first vector, x
    -> Int       -- ^ The increment, incx, between elements drawn from x
    -> Vector a -- ^ The second vector, y
    -> Int       -- ^ The increment, incy,  between elements drawn from y
    -> a         -- ^ The sum of products of the selected elements
{-# INLINE dot #-}
dot !n sx !incx sy !incy = V.foldl' (+) 0
    $ V.zipWith (*) (sampleElems n sx incx) $ sampleElems n sy incy
{-# DEPRECATED sdot,ddot "Use dot instead." #-}
sdot :: Int -> Vector Float -> Int -> Vector Float -> Int -> Float
sdot = dot
ddot :: Int -> Vector Double -> Int -> Vector Double -> Int -> Double
ddot = dot

-- O(1) - Compute the starting index of an iterative vector traversal
-- given the length of the vector and the iterative step size.
firstIndex :: Int -> Int -- ^ the iterative step
           -> Int -- ^ The index of the first iterative step
{-# INLINE firstIndex #-}
firstIndex !n !inc
    | inc>0     = 0
    | otherwise = (1-n)*inc

-- | O(n) - Multiply selected elements in a vector, x, by a scalar, a.
     This
-- function is morally equivalent to scalar multiplication a*x when n is
     the
-- length x and incx is unity.
scal :: (V.Storable a, Num a)
    => Int       -- ^ The number of elements to scale, n
    -> a         -- ^ The scalar factor, a
```

List based implementation is substantially slower than FORTRAN BLAS



Time complexity is linear in n, but linear in inc as well.

Making Sense of the Poor Performance of the List Implementation

OBSERVATIONS

- All three special cases fuse properly to eliminate list traversals.

Prettified Simplifer Output

```
specialCase0 n u v =
    let u0 = head v
        v1 = head u
        z  = u0 * v0
        go i subtotal =
            case i of
                1 -> subtotal
                _ -> go (i-1) (subtotal+z)
    in  go n 0

specialCase1 n u v =
    let go us vs subtotal = case us of
            [] -> subtotal
            x:xs -> case vs
                [] -> subtotal
                y:ys -> case i
                    1 -> subtotal + x*y
                    _ -> go xs ys (i-1) (subtotal + x*y)
    in  go u v n

specialCase2 n u incx v incy =
    let aincy = if incy<0 then negate incy else incy
        aincx = if incx<0 then negate incx else incx
        go i xs ys subtotal = case (i == n) of
            False -> case xs
                [] -> badHead;
                (x:xs') -> case ys of
                    [] -> badHead;
                    (y:ys')-> go (i+1)
                                 (drop aincx xs)
                                 (drop aincy ys)
                                 (subtotal + x*y)
            True -> subtotal
    in go 0 u v 0.0
```

Making Sense of the Poor Performance of the List Implementation

OBSERVATIONS

- All three special cases fuse properly to eliminate list traversals.
- Unnecessary guards against empty lists lead to larger O(n) constant.

Possible Resolution

Eliminating the guards could improve time performance, but calling program would need to ensure validity of arguments.

Prettified Simplifer Output

```
specialCase0 n u v =
    let u0 = head v
        v1 = head u
        z  = u0 * v0
        go i subtotal =
            case i of
                1 -> subtotal
                _ -> go (i-1) (subtotal+z)
    in  go n 0

specialCase1 n u v =
    let go us vs subtotal = case us of
            [] -> subtotal
            x:xs -> case vs
                [] -> subtotal
                y:ys -> case i
                    1 -> subtotal + x*y
                    _ -> go xs ys (i-1) (subtotal + x*y)
    in  go u v n

specialCase2 n u incx v incy =
    let aincy = if incy<0 then negate incy else incy
        aincx = if incx<0 then negate incx else incx
        go i xs ys subtotal = case (i == n) of
            False -> case xs
                [] -> badHead;
                (x:xs') -> case ys of
                    [] -> badHead;
                    (y:ys')-> go (i+1)
                                 (drop aincx xs)
                                 (drop aincy ys)
                                 (subtotal + x*y)
            True -> subtotal
    in go 0 u v 0.0
```

Making Sense of the Poor Performance of the List Implementation

OBSERVATIONS

- All three special cases fuse properly to eliminate list traversals.
- Unnecessary guards against empty lists lead to larger O(n) constant.
- Dereferencing elements from the linked list lead to larger O(n) constant.

Possible Resolution

Choosing a datatype with unboxed elements could reduce dereferencing expense.

Prettified Simplifer Output

```
specialCase0 n u v =
    let u0 = head v
        v1 = head u
        z  = u0 * v0
        go i subtotal =
            case i of
                1 -> subtotal
                _ -> go (i-1) (subtotal+z)
    in  go n 0

specialCase1 n u v =
    let go us vs subtotal = case us of
            [] -> subtotal
            x:xs -> case vs
                [] -> subtotal
                y:ys -> case i
                    1 -> subtotal + x*y
                    _ -> go xs ys (i-1) (subtotal + x*y)
    in  go u v n

specialCase2 n u incx v incy =
    let aincy = if incy<0 then negate incy else incy
        aincx = if incx<0 then negate incx else incx
        go i xs ys subtotal = case (i == n) of
            False -> case xs
                [] -> badHead;
                (x:xs') -> case ys of
                    [] -> badHead;
                    (y:ys')-> go (i+1)
                                 (drop aincx xs)
                                 (drop aincy ys)
                                 (subtotal + x*y)
            True -> subtotal
    in go 0 u v 0.0
```

Making Sense of the Poor Performance of the List Implementation

OBSERVATIONS

- All three special cases fuse properly to eliminate list traversals.
- Unnecessary guards against empty lists lead to larger O(n) constant.
- Dereferencing elements from the linked list lead to larger O(n) constant.
- Desugaring of **abs** into if then else statement leads to larger O(n) constant.

Possible Resolution

A GHC ticket will address this issue (ticket 13212). Bit twiddling could improve time performance in the near term.

Prettified Simplifer Output

```
specialCase0 n u v =
    let u0 = head v
        v1 = head u
        z  = u0 * v0
        go i subtotal =
            case i of
                1 -> subtotal
                _ -> go (i-1) (subtotal+z)
    in  go n 0

specialCase1 n u v =
    let go us vs subtotal = case us of
            [] -> subtotal
            x:xs -> case vs
                [] -> subtotal
                y:ys -> case i
                    1 -> subtotal + x*y
                    _ -> go xs ys (i-1) (subtotal + x*y)
    in  go u v n

specialCase2 n u incx v incy =
    let aincy = if incy<0 then negate incy else incy
        aincx = if incx<0 then negate incx else incx
        go i xs ys subtotal = case (i == n) of
            False -> case xs
                [] -> badHead;
                (x:xs') -> case ys of
                    [] -> badHead;
                    (y:ys')-> go (i+1)
                                 (drop aincx xs)
                                 (drop aincy ys)
                                 (subtotal + x*y)
            True -> subtotal
    in go 0 u v 0.0
```

Making Sense of the Poor Performance of the List Implementation

OBSERVATIONS

- All three special cases fuse properly to eliminate list traversals.
- Unnecessary guards against empty lists lead to larger O(n) constant.
- Dereferencing elements from the linked list lead to larger O(n) constant.
- Desugaring of **abs** into if then else statement leads to larger O(n) constant.
- Dereferencing of dropped cons cells leads to O(inc) time complexity.

Possible Resolution

Using a datatype with O(1) access (e.g. arrays) could dramatically improve the time performance.

Prettified Simplifer Output

```
specialCase0 n u v =
    let u0 = head v
        v1 = head u
        z  = u0 * v0
        go i subtotal =
            case i of
                1 -> subtotal
                _ -> go (i-1) (subtotal+z)
    in  go n 0

specialCase1 n u v =
    let go us vs subtotal = case us of
            [] -> subtotal
            x:xs -> case vs
                [] -> subtotal
                y:ys -> case i
                    1 -> subtotal + x*y
                    _ -> go xs ys (i-1) (subtotal + x*y)
    in  go u v n

specialCase2 n u incx v incy =
    let aincy = if incy<0 then negate incy else incy
        aincx = if incx<0 then negate incx else incx
        go i xs ys subtotal = case (i == n) of
            False -> case xs
                [] -> badHead;
                (x:xs') -> case ys of
                    [] -> badHead;
                    (y:ys')-> go (i+1)
                                 (drop aincx xs)
                                 (drop aincy ys)
                                 (subtotal + x*y)
            True -> subtotal
    in go 0 u v 0.0
```

Making Sense of the Poor Performance of the List Implementation

OBSERVATIONS

- All three special cases fuse properly to eliminate list traversals.
- Unnecessary guards against empty lists lead to larger O(n) constant.
- Dereferencing elements from the linked list lead to larger O(n) constant.
- Desugaring of **abs** into if then else statement leads to larger O(n) constant.
- Dereferencing of dropped cons cells leads to O(inc) time complexity.
- Reversal of lists does not fuse and leads to larger O(inc) and O(n) constants.

Possible Resolution

Using a datatype with O(1) access (e.g. arrays) could dramatically improve the time performance.

Prettified Simplifer Output

```
specialCase0 n u v =
    let u0 = head v
        v1 = head u
        z  = u0 * v0
        go i subtotal =
            case i of
                1 -> subtotal
                _ -> go (i-1) (subtotal+z)
    in  go n 0

specialCase1 n u v =
    let go us vs subtotal = case us of
            [] -> subtotal
            x:xs -> case vs
                [] -> subtotal
                y:ys -> case i
                    1 -> subtotal + x*y
                    _ -> go xs ys (i-1) (subtotal + x*y)
    in  go u v n

specialCase2 n u incx v incy =
    let aincy = if incy<0 then negate incy else incy
        aincx = if incx<0 then negate incx else incx
        go i xs ys subtotal = case (i == n) of
            False -> case xs
                [] -> badHead;
                (x:xs') -> case ys of
                    [] -> badHead;
                    (y:ys')-> go (i+1)
                                 (drop aincx xs)
                                 (drop aincy ys)
                                 (subtotal + x*y)
            True -> subtotal
    in go 0 u v 0.0
```

Improved Implementation : Stream Fusion

FUSION STRATEGY

Lists producing functions are represented as co-recursive data of the *StreamF* bifunctor. (Coutts, Leshchinskiy, and Stewart 2007)

All list consuming functions are expressed as folds. Both right and left associated folds can be implemented with the same efficiency.

Stream pipeline functions can be fused automatically by standard compiler optimizations (S. L. Peyton Jones and Santos 1998).

Generalized stream fusion provides support for SIMD operations (PeytonJones2013).

Mathematical Specification

$$\begin{aligned} \mathbf{StreamF}(A, B) &\mapsto Unit \dot{+} B \dot{+} A \dot{times} B \\ \mathbf{StreamF}\, f, g &\mapsto id \dot{+} g \dot{+} f \dot{times} g \end{aligned}$$

$$CoStream(A) \cong \mathbf{CoChurch}\ (\mathbf{StreamF}\ A)\ A$$

Improved Implementation : Stream Fusion

FUSION STRATEGY

Lists producing functions are represented as co-recursive data of the *StreamF* bifunctor. (Coutts, Leshchinskiy, and Stewart 2007)

All list consuming functions are expressed as folds. Both right and left associated folds can be implemented with the same efficiency.

Stream pipeline functions can be fused automatically by standard compiler optimizations (S. L. Peyton Jones and Santos 1998).

Generalized stream fusion provides support for SIMD operations (**PeytonJones2013**).

IMPLEMENTATION STRATEGY

- Written in Haskell-2010

Haskell Source Code

```
{-# INLINE copy #-}
copy n sx incx sy incy = updateElems (\ _ x -> x ) n sy incy sx incx
{-# DEPRECATED scopy,dcopy "Use copy instead" #-}
scopy :: Int -> Vector Float -> Int -> Vector Float -> Int -> Vector Float
scopy = copy
dcopy :: Int -> Vector Double -> Int -> Vector Double -> Int -> Vector
     Double
dcopy = copy

-- |  O(n) -- Swap n elements between two Float vectors.
swap :: (V.Storable a)
    => Int       -- ^ The number of elements to copy
    -> Vector a -- ^ The first vector from which elements will be swapped
    -> Int       -- ^ The increment between elements drawn from the first
         vector
    -> Vector a -- ^ The second vector from which elements will be swapped
    -> Int       -- ^ The increment between elements drawn from the second
         vector
    -> (Vector a, Vector a)
{-# INLINE swap #-}
swap n sx incx sy incy =
    ( updateElems (\ _ y -> y ) n sx incx sy incy
    , updateElems (\ _ x -> x ) n sy incy sx incx )
{-# DEPRECATED sswap,dswap "Use swap instead." #-}
sswap :: Int -> Vector Float -> Int -> Vector Float -> Int -> (Vector Float
     , Vector Float)
sswap = swap
dswap :: Int -> Vector Double -> Int -> Vector Double -> Int -> (Vector
     Double, Vector Double)
```



Improved Implementation : Stream Fusion

FUSION STRATEGY

Lists producing functions are represented as co-recursive data of the *StreamF* bifunctor. (Coutts, Leshchinskiy, and Stewart 2007)

All list consuming functions are expressed as folds. Both right and left associated folds can be implemented with the same efficiency.

Stream pipeline functions can be fused automatically by standard compiler optimizations (S. L. Peyton Jones and Santos 1998).

Generalized stream fusion provides support for SIMD operations (**PeytonJones2013**).

IMPLEMENTATION STRATEGY

- Written in Haskell-2010
- Vectors are represented using the Data.Vector.Storable module, version 0.11.0.0.

Haskell Source Code

```
{-# INLINE copy #-}
copy n sx incx sy incy = updateElems (\ _ x -> x ) n sy incy sx incx
{-# DEPRECATED scopy,dcopy "Use copy instead" #-}
scopy :: Int -> Vector Float -> Int -> Vector Float -> Int -> Vector Float
scopy = copy
dcopy :: Int -> Vector Double -> Int -> Vector Double -> Int -> Vector
     Double
dcopy = copy

-- |  O(n) -- Swap n elements between two Float vectors.
swap :: (V.Storable a)
    => Int       -- ^ The number of elements to copy
    -> Vector a -- ^ The first vector from which elements will be swapped
    -> Int       -- ^ The increment between elements drawn from the first
         vector
    -> Vector a -- ^ The second vector from which elements will be swapped
    -> Int       -- ^ The increment between elements drawn from the second
         vector
    -> (Vector a, Vector a)
{-# INLINE swap #-}
swap n sx incx sy incy =
    ( updateElems (\ _ y -> y ) n sx incx sy incy
    , updateElems (\ _ x -> x ) n sy incy sx incx )
{-# DEPRECATED sswap,dswap "Use swap instead." #-}
sswap :: Int -> Vector Float -> Int -> Vector Float -> Int -> (Vector Float
     , Vector Float)
sswap = swap
dswap :: Int -> Vector Double -> Int -> Vector Double -> Int -> (Vector
     Double, Vector Double)
```



Improved Implementation : Stream Fusion

FUSION STRATEGY

Lists producing functions are represented as co-recursive data of the *StreamF* bifunctor. (Coutts, Leshchinskiy, and Stewart 2007)

All list consuming functions are expressed as folds. Both right and left associated folds can be implemented with the same efficiency.

Stream pipeline functions can be fused automatically by standard compiler optimizations (S. L. Peyton Jones and Santos 1998).

Generalized stream fusion provides support for SIMD operations (**PeytonJones2013**).

IMPLEMENTATION STRATEGY

- Written in Haskell-2010
- Vectors are represented using the Data.Vector.Storable module, version 0.11.0.0.
- O(n) When both inc_x and inc_y are zero, compute the solution as a fold/replicate composite.

Haskell Source Code

```
{-# INLINE copy #-}
copy n sx incx sy incy = updateElems (\ _ x -> x ) n sy incy sx incx
{-# DEPRECATED scopy,dcopy "Use copy instead" #-}
scopy :: Int -> Vector Float -> Int -> Vector Float -> Int -> Vector Float
scopy = copy
dcopy :: Int -> Vector Double -> Int -> Vector Double -> Int -> Vector
     Double
dcopy = copy

-- |  O(n) -- Swap n elements between two Float vectors.
swap :: (V.Storable a)
    => Int       -- ^ The number of elements to copy
    -> Vector a -- ^ The first vector from which elements will be swapped
    -> Int       -- ^ The increment between elements drawn from the first
         vector
    -> Vector a -- ^ The second vector from which elements will be swapped
    -> Int       -- ^ The increment between elements drawn from the second
         vector
    -> (Vector a, Vector a)
{-# INLINE swap #-}
swap n sx incx sy incy =
    ( updateElems (\ _ y -> y ) n sx incx sy incy
    , updateElems (\ _ x -> x ) n sy incy sx incx )
{-# DEPRECATED sswap,dswap "Use swap instead." #-}
sswap :: Int -> Vector Float -> Int -> Vector Float -> Int -> (Vector Float
     , Vector Float)
sswap = swap
dswap :: Int -> Vector Double -> Int -> Vector Double -> Int -> (Vector
     Double, Vector Double)
```

Improved Implementation : Stream Fusion

FUSION STRATEGY

Lists producing functions are represented as co-recursive data of the *StreamF* bifunctor. (Coutts, Leshchinskiy, and Stewart 2007)

All list consuming functions are expressed as folds. Both right and left associated folds can be implemented with the same efficiency.

Stream pipeline functions can be fused automatically by standard compiler optimizations (S. L. Peyton Jones and Santos 1998).

Generalized stream fusion provides support for SIMD operations (**PeytonJones2013**).

IMPLEMENTATION STRATEGY

- Written in Haskell-2010
- Vectors are represented using the Data.Vector.Storable module, version 0.11.0.0.
- O(n) When both inc_x and inc_y are zero, compute the solution as a fold/replicate composite.
- O(n) When one of the increments is zero, compute the solution as a fold/map composite.

Haskell Source Code

```
{-# INLINE copy #-}
copy n sx incx sy incy = updateElems (\ _ x -> x ) n sy incy sx incx
{-# DEPRECATED scopy,dcopy "Use copy instead" #-}
scopy :: Int -> Vector Float -> Int -> Vector Float -> Int -> Vector Float
scopy = copy
dcopy :: Int -> Vector Double -> Int -> Vector Double -> Int -> Vector
    Double
dcopy = copy

-- |  O(n) -- Swap n elements between two Float vectors.
swap :: (V.Storable a)
    => Int       -- ^ The number of elements to copy
    -> Vector a -- ^ The first vector from which elements will be swapped
    -> Int       -- ^ The increment between elements drawn from the first
        vector
    -> Vector a -- ^ The second vector from which elements will be swapped
    -> Int       -- ^ The increment between elements drawn from the second
        vector
    -> (Vector a, Vector a)
{-# INLINE swap #-}
swap n sx incx sy incy =
    ( updateElems (\ _ y -> y ) n sx incx sy incy
    , updateElems (\ _ x -> x ) n sy incy sx incx )
{-# DEPRECATED sswap,dswap "Use swap instead." #-}
sswap :: Int -> Vector Float -> Int -> Vector Float -> Int -> (Vector Float
    , Vector Float)
sswap = swap
dswap :: Int -> Vector Double -> Int -> Vector Double -> Int -> (Vector
    Double, Vector Double)
```

Improved Implementation : Stream Fusion

FUSION STRATEGY

Lists producing functions are represented as co-recursive data of the *StreamF* bifunctor. (Coutts, Leshchinskiy, and Stewart 2007)

All list consuming functions are expressed as folds. Both right and left associated folds can be implemented with the same efficiency.

Stream pipeline functions can be fused automatically by standard compiler optimizations (S. L. Peyton Jones and Santos 1998).

Generalized stream fusion provides support for SIMD operations (PeytonJones2013).

IMPLEMENTATION STRATEGY

- Written in Haskell-2010
- Vectors are represented using the Data.Vector.Storable module, version 0.11.0.0.
- O(n) When both inc_x and inc_y are zero, compute the solution as a fold/replicate composite.
- O(n) When one of the increments is zero, compute the solution as a fold/map composite.
- O(n) For all other cases, express the solution as a fold/zipWith composite.

Haskell Source Code

```
{-# INLINE copy #-}
copy n sx incx sy incy = updateElems (\ _ x -> x ) n sy incy sx incx
{-# DEPRECATED scopy,dcopy "Use copy instead" #-}
scopy :: Int -> Vector Float -> Int -> Vector Float -> Int -> Vector Float
scopy = copy
dcopy :: Int -> Vector Double -> Int -> Vector Double -> Int -> Vector
    Double
dcopy = copy

-- |  O(n) -- Swap n elements between two Float vectors.
swap :: (V.Storable a)
    => Int       -- ^ The number of elements to copy
    -> Vector a -- ^ The first vector from which elements will be swapped
    -> Int       -- ^ The increment between elements drawn from the first
        vector
    -> Vector a -- ^ The second vector from which elements will be swapped
    -> Int       -- ^ The increment between elements drawn from the second
        vector
    -> (Vector a, Vector a)
{-# INLINE swap #-}
swap n sx incx sy incy =
    ( updateElems (\ _ y -> y ) n sx incx sy incy
    , updateElems (\ _ x -> x ) n sy incy sx incx )
{-# DEPRECATED sswap,dswap "Use swap instead." #-}
sswap :: Int -> Vector Float -> Int -> Vector Float -> Int -> (Vector Float
    , Vector Float)
sswap = swap
dswap :: Int -> Vector Double -> Int -> Vector Double -> Int -> (Vector
    Double, Vector Double)
```

Improved Implementation : Stream Fusion

FUSION STRATEGY

Lists producing functions are represented as co-recursive data of the *StreamF* bifunctor. (Coutts, Leshchinskiy, and Stewart 2007)

All list consuming functions are expressed as folds. Both right and left associated folds can be implemented with the same efficiency.

Stream pipeline functions can be fused automatically by standard compiler optimizations (S. L. Peyton Jones and Santos 1998).

Generalized stream fusion provides support for SIMD operations (PeytonJones2013).

IMPLEMENTATION STRATEGY

- Written in Haskell-2010
- Vectors are represented using the Data.Vector.Storable module, version 0.11.0.0.
- O(n) When both inc_x and inc_y are zero, compute the solution as a fold/replicate composite.
- O(n) When one of the increments is zero, compute the solution as a fold/map composite.
- O(n) For all other cases, express the solution as a fold/zipWith composite.
- O(n) Elements of the vectors are extracted using sample and samplerev, eliminating the need for drop/take.

Haskell Source Code

```
{-# INLINE copy #-}
copy n sx incx sy incy = updateElems (\ _ x -> x ) n sy incy sx incx
{-# DEPRECATED scopy,dcopy "Use copy instead" #-}
scopy :: Int -> Vector Float -> Int -> Vector Float -> Int -> Vector Float
scopy = copy
dcopy :: Int -> Vector Double -> Int -> Vector Double -> Int -> Vector
     Double
dcopy = copy

-- |  O(n) -- Swap n elements between two Float vectors.
swap :: (V.Storable a)
    => Int       -- ^ The number of elements to copy
    -> Vector a -- ^ The first vector from which elements will be swapped
    -> Int       -- ^ The increment between elements drawn from the first
         vector
    -> Vector a -- ^ The second vector from which elements will be swapped
    -> Int       -- ^ The increment between elements drawn from the second
         vector
    -> (Vector a, Vector a)
{-# INLINE swap #-}
swap n sx incx sy incy =
    ( updateElems (\ _ y -> y ) n sx incx sy incy
    , updateElems (\ _ x -> x ) n sy incy sx incx )
{-# DEPRECATED sswap,dswap "Use swap instead." #-}
sswap :: Int -> Vector Float -> Int -> Vector Float -> Int -> (Vector Float
     , Vector Float)
sswap = swap
dswap :: Int -> Vector Double -> Int -> Vector Double -> Int -> (Vector
     Double, Vector Double)
```

Improved Implementation : Stream Fusion

FUSION STRATEGY

Lists producing functions are represented as co-recursive data of the *StreamF* bifunctor. (Coutts, Leshchinskiy, and Stewart 2007)

All list consuming functions are expressed as folds. Both right and left associated folds can be implemented with the same efficiency.

Stream pipeline functions can be fused automatically by standard compiler optimizations (S. L. Peyton Jones and Santos 1998).

Generalized stream fusion provides support for SIMD operations (PeytonJones2013).

IMPLEMENTATION STRATEGY

- Written in Haskell-2010
- Vectors are represented using the Data.Vector.Storable module, version 0.11.0.0.
- O(n) When both inc_x and inc_y are zero, compute the solution as a fold/replicate composite.
- O(n) When one of the increments is zero, compute the solution as a fold/map composite.
- O(n) For all other cases, express the solution as a fold/zipWith composite.
- O(n) Elements of the vectors are extracted using sample and samplerev, eliminating the need for drop/take.
- Calling program assumes responsibility for ensuring vectors are long enough.

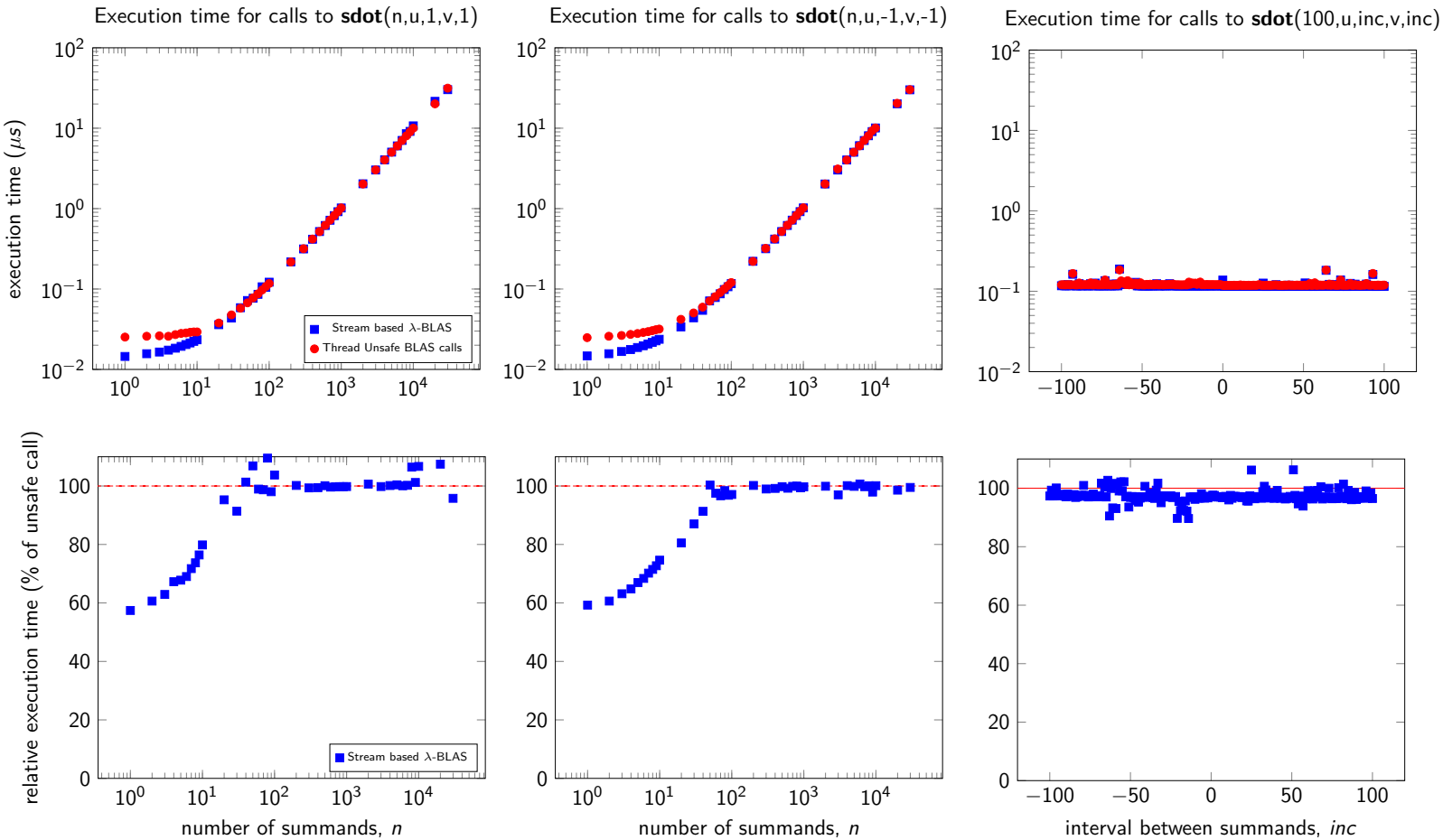
Haskell Source Code

```
{-# INLINE copy #-}
copy n sx incx sy incy = updateElems (\ _ x -> x ) n sy incy sx incx
{-# DEPRECATED scopy,dcopy "Use copy instead" #-}
scopy :: Int -> Vector Float -> Int -> Vector Float -> Int -> Vector Float
scopy = copy
dcopy :: Int -> Vector Double -> Int -> Vector Double -> Int -> Vector
    Double
dcopy = copy

-- |  O(n) -- Swap n elements between two Float vectors.
swap :: (V.Storable a)
    => Int       -- ^ The number of elements to copy
    -> Vector a -- ^ The first vector from which elements will be swapped
    -> Int       -- ^ The increment between elements drawn from the first
        vector
    -> Vector a -- ^ The second vector from which elements will be swapped
    -> Int       -- ^ The increment between elements drawn from the second
        vector
    -> (Vector a, Vector a)
{-# INLINE swap #-}
swap n sx incx sy incy =
    ( updateElems (\ _ y -> y ) n sx incx sy incy
    , updateElems (\ _ x -> x ) n sy incy sx incx )
{-# DEPRECATED sswap,dswap "Use swap instead." #-}
sswap :: Int -> Vector Float -> Int -> Vector Float -> Int -> (Vector Float
    , Vector Float)
sswap = swap
dswap :: Int -> Vector Double -> Int -> Vector Double -> Int -> (Vector
    Double, Vector Double)
```

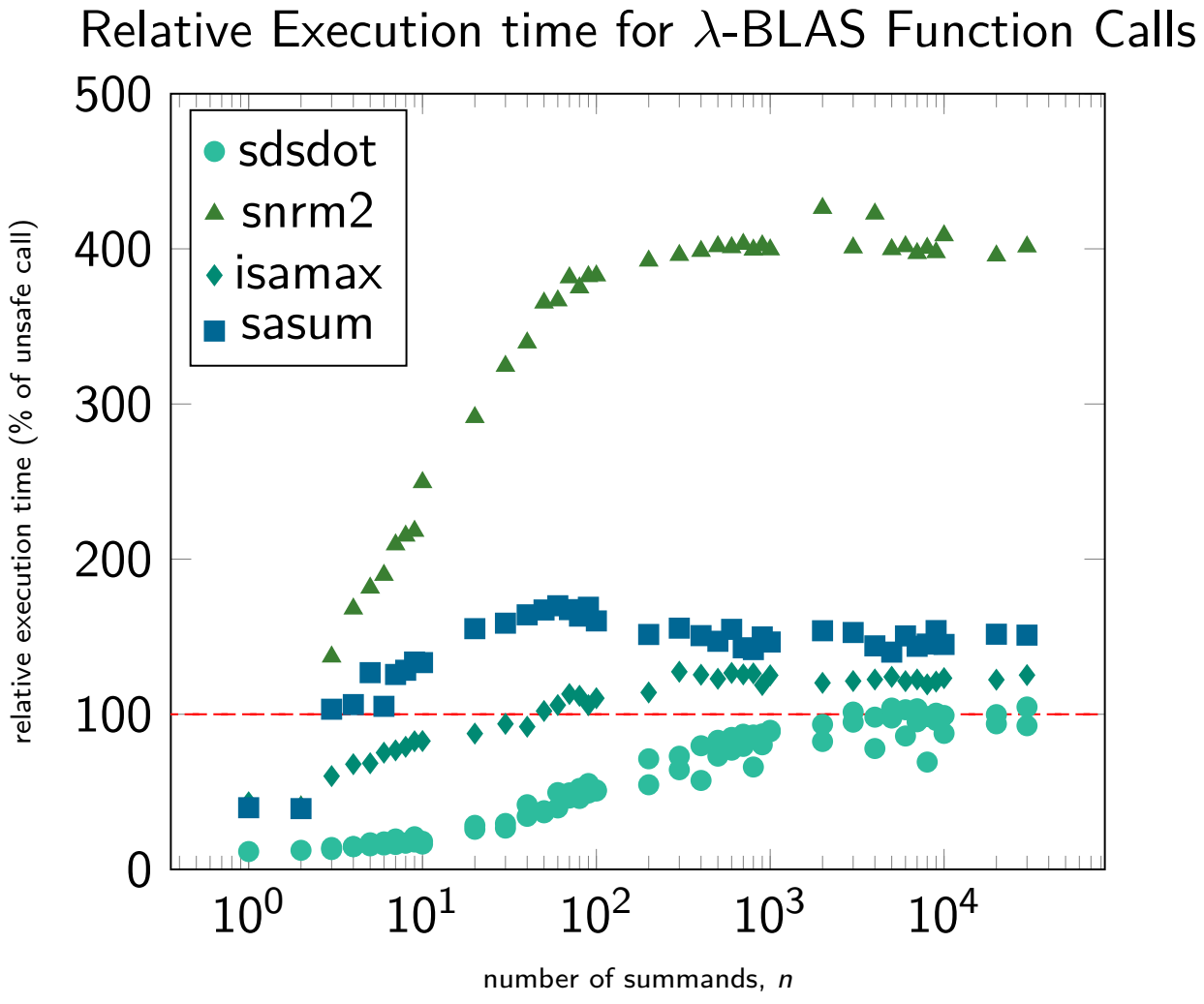


Vector Based Implementation has Comparable or Better Performance than FORTRAN BLAS



Time complexity is linear in n and constant in inc.

Implementation of other Level-1 BLAS Functions is In Progress and Shows Promise



Section 5

Further Work

- Complete implementation of BLAS level-1 functions
- Static checking of vector bounds
- Examine options and tradeoffs for upstream fusion.
- Explore concurrent implementations of BLAS functions
- Improved precision (e.g. Kahan summation, cascading summation)
- Release into the wild



Section 6

Conclusions

- It is possible to generate beautiful numerical methods that are as fast as FORTRAN.
- Generalized stream fusion works well for optimization of vector based algorithms.
- Haskell's rewrite rule system facilitates writing high level, conceptually clear and performant code.



Acknowledgements

David Balaban	Amgen, Inc
Nicholas Frisby	Amgen Inc
Gary Howard	Allen Consulting
David Spivak	MIT Department of Mathematics

References

- Bird, Richard (1988). *Introduction to Functional Programming In Haskell*. Ed. by C.A.R. Hoare. Prentice Hall.
- Coutts, Duncan, Roman Leshchinskiy, and Don Stewart (2007). "Stream Fusion. From Lists to Streams to Nothing at All". In: *Proceedings of the 2007 ACM SIGPLAN international conference on Functional programming - ICFP '07* 42.9, p. 315.
- Fokkinga, Maarten M (1994). *A Gentle Introduction to Category Theory*. Tech. rep. Utrecht: University of Utrecht, p. 80.
- Fokkinga, Maarten M. and Erik Meijer (Jan. 1991). *Program Calculation Properties of Continuous Algebras*.
- Gibbons, Jeremy (2007). *Datatype-Generic Programming*.
- — (2008). "Unfolding abstract datatypes". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5133 LNCS, pp. 110–133.
- Gill, Andrew, John Launchbury, and Simon Peyton Jones (1993). "A short cut to deforestation". In: *Proceedings of the conference on Functional programming languages and computer architecture - FPCA '93* Section 4, pp. 223–232.
- Mainland, Geoffrey, Roman Leshchinskiy, and Simon Peyton Jones (2013). "Exploiting Vector Instructions with Generalized Stream Fusion". In: *the 18th ACM SIGPLAN international conference* Section 4, p. 37.
- Meertens, Lambert (1986). "Algorithmics – Towards programming as a mathematical activity". In: pp. 289–334.
- Meijer, Erik, Maarten Fokkinga, and Ross Paterson (1991). "Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire". In: *Functional Programming Languages and Computer Architecture*, pp. 124–144.
- Perlis, Alan (1982). "Epigrams in Programming". In: *ACM SIGPLAN Notices* 17 (9), pp. 7–13.
- Peyton Jones, Simon, Andrew Tolmach, and Tony Hoare (Sept. 2001). "Playing by the rules: rewriting as a practical optimisation technique in GHC". In.
- Peyton Jones, Simon L. and André L. M. Santos (1998). "A Transformation-based Optimiser for Haskell". In: *Sci. Comput. Program.* 32.1-3, pp. 3–47.
- Richard Bird, Oege De Moor (1997). *The Algebra of Programming*. Harlow, England: Pearson.
- Wadler, Philip (1990). *Recursive Types For Free*.

