1 Introduction

OptiML [9] is a domain-specific language for machine learning. It is embedded in Scala, and built on top of the Lightweight Modular Staging (LMS) [7] and Delite [1] frameworks. This document describes the structure, API and restrictions for OptiML [9] programs. It is organized into several categories. Section 2 describes how an OptiML program is structured, and its basic operations, which are a subset of Scala. Section 4 and section 5 respectively describe the OptiML data and control structures and their associated operations and semantics. Section 6 describes the remaining built-in language operators. Section 7 describes the most important optimizations performed by the OptiML compiler. Finally, section 8 discusses some more subtle issues related to programming in OptiML and section 9 discusses off-the-shelf machine learning algorithms available in the standard library.

Before diving in the details, it is important to clarify our design goals. OptiML is currently targeted at machine learning researchers and algorithm developers; it aims to provide a productive, high performance, MATLAB-like environment for linear algebra supplemented with machine learning specific abstractions. Our primary goal is to allow machine learning practitioners to write code in a highly declarative manner and still achieve high performance on a variety of underlying parallel, heterogeneous devices. The same OptiML program should run well and scale on a CMP (chip multi-processor), a GPU, a combination of CMPs and GPUs, clusters of CMPs and GPUs, and eventually even FPGAs and other specialized accelerators.

In particular, OptiML is designed to allow statistical inference algorithms expressible by the Statistical Query Model [3] to be both easy to express and very fast to execute. These algorithms can be expressed in a summation form [2], and can be parallelized using fine-grained map-reduce operations. OptiML employs aggressive optimizations to reduce unnecessary memory allocations and fuse operations together to make these as fast as possible. OptiML also attempts to specialize implementations to particular hardware devices as much as possible to achieve the best performance.

Although nascent, OptiML also includes the beginning of a standard library for ML that incorporates common machine learning algorithms and classifiers. These functions are written in OptiML and allow machine learning application developers to easily make use of off-the-shelf machine learning algorithms.
2 Basics

2.1 Program structure

OptiML applications have the following basic structure:

```scala
import ppl.dsl.optiml._

object MyApplicationRunner extends OptiMLApplicationRunner with MyApplication

trait MyApplication extends OptiMLApplication {
  def main() = { ... }
}
```

The division into `Application` and `ApplicationRunner` is due to the fact that OptiML is embedded in Scala using LMS; for more details about why this is necessary, see [8].

`main()` is the entry-point for OptiML applications. It implicitly imports an `args` variable into scope, that can be used to access parameters supplied to the program at run-time. For example, the command

```
delite MyApplicationRunner "hello"
```

would allow the parameter "hello" to be accessed using `args(0)`.

2.2 Declaring values and variables

The keywords `val` and `var` are used to declare constant and variable identifiers respectively. For example,

```
val x = 100
var y = 100
x = 7 // error; vals cannot be modified
y = 7 // ok
```

2.3 Conditionals

OptiML supports standard if/then/else statements. The syntax is:

```
if (cond) { ... }
else if { ... }
else { ... }
```

where any of the else clauses may be omitted.

2.4 Sequential iteration

OptiML supports `while` loops in the usual way. The semantics of OptiML’s while loops are the same as Scala, Java, and C++. In particular, each iteration is guaranteed to be sequentially consistent. Note that

---

1 OptiML does not support Scala ranges, so standard Scala 'for loops' will generate a type error.
OptiML will never attempt to parallelize across while loop iterations, but may parallelize independent tasks within an iteration. The result of a while loop is always deterministic. Example usage:

```scala
var i = 0
while (i < 10) {
    ...
}
```

### 2.5 Methods

OptiML allows methods to be defined using the `def` keyword. For example,

```scala
def main() = {
    val x = foo
    val y = bar()
    println(y) // prints "bar"
}
def foo() = {
    "foo"
}
def bar() = {
    "bar"
}
```

This example defines two (pointless) methods that are invoked from inside `main`. Note that like Scala, parantheses may be omitted when calling a method. However, OptiML methods have two major differences from Scala methods:

- **return** is not supported. While this will eventually be rejected or implemented, currently it results in undefined behavior. The value on the last line of the method is the return value for the method, as in the previous examples.

- method parameters must be wrapped with the `Rep` type.

Here is an example of using wrapped method parameters:

```scala
def foo(val x: Rep[Int], val y: Rep[Double], val z: Rep[Vector[Double]]) { ... }
```

### 2.6 Utility operations

OptiML provides a small set of utility operations automatically available in application scope:

- `print, println`

- `exit`

---

2 This is an example where details of the embedding implementation currently still leak into application code. We are working to make this unnecessary with newer versions of the virtualized Scala compiler and LMS library.
2.7 Scala subsetting

The astute reader familiar with Scala will notice that the set of operations listed here are a small subset of Scala operations and syntax. Any Scala operations or keywords not listed here are unsupported, and will result in undefined behavior (usually type errors). In the future, we intend to make this more robust, so that unsupported Scala operations are explicitly rejected with a sensible error message.

3 Restricted Functions

OptiML has a number of operations that accept user-defined functions. In order to effectively fuse and parallelize these operations, OptiML does not allow arbitrary functions, but instead usually requires restricted functions. An OptiML restricted function is a function that must be pure, i.e. contain no side-effects. If an operation expecting a restricted function is supplied a function with arbitrary side-effects, it will (eventually) be rejected by the OptiML compiler during staging (this is not currently implemented). Certain OptiML operations relax this restriction in structured ways, and allow mutations to certain elements within a data structure; these relaxations are specified on a per-operation basis.

4 Data Types

OptiML supports a small number of data types geared towards linear algebra and machine learning. While it is possible to use import statements to import other Scala types into scope, these are not legal to use inside OptiML programs, and will usually result in type errors.

The data types supported by OptiML can be organized into several categories: primitives, vectors, matrices, streams, and graphs. All non-primitive OptiML data types are polymorphic and can be used with any type parameter. In other words, you can instantiate a DenseVector[Foo] if the type Foo exists in the OptiML application. One practical use-case for this is vectors which contain other vectors.

All OptiML data types support the following methods:

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsInstanceOf</td>
<td>dynamic type test</td>
</tr>
<tr>
<td>AsInstanceOf</td>
<td>type cast</td>
</tr>
<tr>
<td>ToString</td>
<td>string representation</td>
</tr>
</tbody>
</table>

Table 1: Generic operations

4.1 Primitives

4.1.1 Numerics

Double, Float, Int, are supported with the following operations:

---

3Note that these are methods that are also built into all Scala objects. They are distinguished in OptiML by the initial capital letter. This discrepancy will be addressed in future versions of the Scala virtualized compiler.
A. Sujeeth

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
</tr>
<tr>
<td>abs</td>
<td>absolute value</td>
</tr>
<tr>
<td>exp</td>
<td>base e exponentiation</td>
</tr>
</tbody>
</table>

Table 2: Numeric operations

### 4.1.2 Boolean

The Boolean type supports:

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>unary negation</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Boolean operations

### 4.1.3 String

The String type supports:

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>string concatenation</td>
</tr>
<tr>
<td>trim</td>
<td>returns a new string with whitespace removed</td>
</tr>
<tr>
<td>split</td>
<td>returns an array of strings</td>
</tr>
<tr>
<td>startsWith(prefix)</td>
<td>tests if the string starts with the given prefix</td>
</tr>
<tr>
<td>toDouble</td>
<td>converts the string to a double</td>
</tr>
<tr>
<td>toFloat</td>
<td>converts the string to a float</td>
</tr>
<tr>
<td>toInt</td>
<td>converts the string to an int</td>
</tr>
</tbody>
</table>

Table 4: String operations

### 4.1.4 Tuple

Tuple can be used to pack multiple values into a single object, and arithmetic on Tuples is supported if all of its members also support arithmetic. OptiML currently supports tuples of up to 5 elements. Below we show an example of using a tuple:

```scala
// pack
val x = (DenseVector.zeros(10), DenseVector.ones(10))
// extract one member
val a = x._1 * 10
// member-wise arithmetic on the tuple
val b = x*2
// extract both of the new result
```
\texttt{val} (zero, twos) = b

### 4.1.5 Function

The Function type allows using functions as values. Functions take 1 or more arguments and return 1 result (which may be \texttt{Unit}, i.e. void). Currently only functions up to 5 arguments are allowed. For example,

\begin{verbatim}
val newVector: Rep[Int] => Rep[DenseVector[Int]] = n => DenseVector[Int](n)
val vecFive = newVector(5)
\end{verbatim}

### 4.2 Vector

Vector represents a generic, linear collection in OptiML. When a Vector is used with a type that supports arithmetic operations (e.g. Double), it is considered a mathematical vector, and possesses the corresponding arithmetic operations (see section \[8.6\]). OptiML distinguishes \texttt{Vector} from \texttt{Matrix} to maintain additional type safety, readability, and optimization opportunities.

There are a number of static operations available for Vector, listed below. These operations do not require a Vector instance, and are used by calling \texttt{Vector.<method>}.

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{Vector(a,b,c,..)}</td>
<td>return a new dense vector containing elements a,b,c,..</td>
</tr>
<tr>
<td>\texttt{Vector.dense(len, isRow)}</td>
<td>return a new empty dense vector of initial size len and orientation based on boolean isRow</td>
</tr>
<tr>
<td>\texttt{Vector.sparse(len, isRow)}</td>
<td>return a new empty sparse vector of initial size len and orientation based on boolean isRow</td>
</tr>
<tr>
<td>ones(len)</td>
<td>return a vector of all double values 1.0</td>
</tr>
<tr>
<td>onesf(len)</td>
<td>return a vector of all float values 1.0</td>
</tr>
<tr>
<td>zeros(len)</td>
<td>return a vector of all double values 0.0</td>
</tr>
<tr>
<td>rand(len)</td>
<td>return a vector of random double values, uniformly distributed from 0 to 1</td>
</tr>
<tr>
<td>randf(len)</td>
<td>return a vector of random float values, uniformly distributed from 0 to 1</td>
</tr>
<tr>
<td>range(start,end,stride)</td>
<td>return a RangeVector containing values from start to end at stride intervals</td>
</tr>
<tr>
<td>uniform(start,end,stride)</td>
<td>return a DenseVector containing double values from start to end at stride intervals</td>
</tr>
</tbody>
</table>

Table 5: Vector static operations

A full description of the operations available on Vector instances follows:

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>toBoolean</td>
<td>converts \texttt{v} to boolean if an element-wise conversion exists</td>
</tr>
<tr>
<td>toDouble</td>
<td>converts \texttt{v} to double if an element-wise conversion exists</td>
</tr>
<tr>
<td>toFloat</td>
<td>converts \texttt{v} to float if an element-wise conversion exists</td>
</tr>
<tr>
<td>toInt</td>
<td>converts \texttt{v} to int if an element-wise conversion exists</td>
</tr>
<tr>
<td>v(i)</td>
<td>returns elements with indices in \texttt{i}</td>
</tr>
<tr>
<td>v(i) = x</td>
<td>set elements in \texttt{i} to value \texttt{x}</td>
</tr>
<tr>
<td>length</td>
<td>returns the length of \texttt{v}</td>
</tr>
<tr>
<td>isRow</td>
<td>true if \texttt{v} is a row vector</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>isEmpty</td>
<td>true if v is empty</td>
</tr>
<tr>
<td>first</td>
<td>returns the first element of v</td>
</tr>
<tr>
<td>last</td>
<td>returns the last element of v</td>
</tr>
<tr>
<td>indices</td>
<td>IndexVector from 0 to length</td>
</tr>
<tr>
<td>drop(c)</td>
<td>returns new vector without the first c elements of v</td>
</tr>
<tr>
<td>take(c)</td>
<td>returns new vector with only the first c elements of v</td>
</tr>
<tr>
<td>slice(s,e)</td>
<td>returns new vector with elements from s to e</td>
</tr>
<tr>
<td>contains(y)</td>
<td>true if y is in v</td>
</tr>
<tr>
<td>distinct</td>
<td>returns new vector with only unique elements in v</td>
</tr>
<tr>
<td>transpose</td>
<td>in-place transpose (modifies v)</td>
</tr>
<tr>
<td>Clone</td>
<td>return an immutable deep-copy</td>
</tr>
<tr>
<td>mutable</td>
<td>return a mutable deep-copy</td>
</tr>
<tr>
<td>pprint</td>
<td>pretty-print v</td>
</tr>
<tr>
<td>replicate(i,j)</td>
<td>matrix with i,j tiles of v</td>
</tr>
<tr>
<td>mkString(sep)</td>
<td>return a string formed by joining all the elements in v with the 'sep' string</td>
</tr>
<tr>
<td>&lt;&lt;(e)</td>
<td>return new vector with appended element e</td>
</tr>
<tr>
<td>&lt;&lt;==(e)</td>
<td>append element e to v in place</td>
</tr>
<tr>
<td>&lt;&lt;==(v2)</td>
<td>return new vector with appended vector v2</td>
</tr>
<tr>
<td>&lt;&lt;==(v2)</td>
<td>append vector v2 to v in place</td>
</tr>
<tr>
<td>copyFrom(p,v2)</td>
<td>overwrite elements of v at index p with elements from v2</td>
</tr>
<tr>
<td>insert(p,e)</td>
<td>insert element e into v at index p, in place</td>
</tr>
<tr>
<td>insertAll(p,v2)</td>
<td>insert all elements from v2 into v at index p, in place</td>
</tr>
<tr>
<td>remove(p)</td>
<td>remove the element at index p from v, in place</td>
</tr>
<tr>
<td>removeAll(p, len)</td>
<td>remove len elements starting at index p from v, in place</td>
</tr>
<tr>
<td>trim</td>
<td>trim the backing store of v to its logical length</td>
</tr>
<tr>
<td>clear</td>
<td>removes all elements from v, in place</td>
</tr>
<tr>
<td>+(v2)</td>
<td>point-wise addition</td>
</tr>
<tr>
<td>+=(v2)</td>
<td>point-wise addition in place</td>
</tr>
<tr>
<td>+=(e)</td>
<td>point-wise addition of e to every element of v in place</td>
</tr>
<tr>
<td>*(v2)</td>
<td>point-wise multiplication</td>
</tr>
<tr>
<td>*=(v2)</td>
<td>point-wise multiplication in place</td>
</tr>
<tr>
<td>*=(e)</td>
<td>point-wise multiplication of e to every element of v in place</td>
</tr>
<tr>
<td>-(v2)</td>
<td>point-wise subtraction</td>
</tr>
<tr>
<td>-==(v2)</td>
<td>point-wise subtraction in place</td>
</tr>
<tr>
<td>-==(e)</td>
<td>point-wise subtraction of e to every element of v in place</td>
</tr>
<tr>
<td>/(v2)</td>
<td>point-wise division</td>
</tr>
<tr>
<td>/==(v2)</td>
<td>point-wise division in place</td>
</tr>
<tr>
<td>/==(e)</td>
<td>point-wise division of e to every element of v in place</td>
</tr>
<tr>
<td>*(m)</td>
<td>vector-matrix product</td>
</tr>
<tr>
<td>**(v2)</td>
<td>outer product</td>
</tr>
<tr>
<td><em>:</em>=(v2)</td>
<td>dot product</td>
</tr>
<tr>
<td>sum</td>
<td>sum of all elements in v</td>
</tr>
<tr>
<td>abs</td>
<td>point-wise absolute value</td>
</tr>
<tr>
<td>exp</td>
<td>point-wise exponentiation</td>
</tr>
<tr>
<td>sort</td>
<td>sorted vector v</td>
</tr>
</tbody>
</table>
min | return minimum value in v
minIndex | return the index of the minimum value in v
max | return maximum value in v
maxIndex | return the index of the maximum value in v
median | return the median value in v
>:>(v2) | return boolean vector representing point-wise > comparison of v and v2
:<(v2) | return boolean vector representing point-wise < comparison of v and v2
map(f) | transform v with a function f†
mmap(f) | transform v in place with a function f†
zip(v2,f) | return a new vector as the result of applying f pair-wise to every element in v and v2†
mzip(v2,f) | transform v in place by applying f pair-wise to every element in v and v2†
reduce(f) | flatten v with a reducing function f†
filter(f) | filter v with a comparator function f†
find(f) | return an index vector containing the indices of elements matching the predicate f†
count(f) | return the number of elements in v matching the predicate f†
flatMap(f) | map each element in v to a Vector using f and concatenate the results†
partition(f) | split v into two vectors based on a predicate f†
groupBy(f) | return a dense vector of vectors, where each inner vector contains elements that map to the same key using function f†

Table 6: Vector class operations

†Must be restricted functions (see Section 3)

4.2.1 DenseVector

The default Vector in OptiML. A contiguous, array-backed sequence of values with an orientation (based on the isRow flag). The DenseVector object provides the following additional methods:

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DenseVector(len, isRow)</td>
<td>construct a new empty dense vector of initial size len and orientation based on boolean isRow equivalent to Vector.dense(len,isRow)</td>
</tr>
<tr>
<td>flatten(vectors)</td>
<td>concatenates a vector of vectors into a single vector</td>
</tr>
</tbody>
</table>

Table 7: DenseVector static operations

4.2.2 SparseVector

A vector that physically stores only non-zero elements. OptiML’s sparse vector is implemented as a coordinate list (two sorted arrays containing non-zero indices and values respectively). The SparseVector object provides the following additional methods:
SparseVector instances also support additional methods:

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nnz</td>
<td>the number of non-zero values in v</td>
</tr>
<tr>
<td>mapNZ(f)</td>
<td>transform v by applying a function f to each non-zero element†</td>
</tr>
<tr>
<td></td>
<td>the resulting vector is also sparse and has the same size (logical and nnz) as v</td>
</tr>
</tbody>
</table>

Table 9: SparseVector class additional operations

4.2.3 RangeVector

A Vector that does not store any actual data, but represents a consecutive range of integers. RangeVector is read-only.

4.2.4 IndexVector

IndexVector is a Vector of non-negative integers representing indices. An IndexVector can represent either a contiguous range (in which case it can be instantiated using the syntax (0::n), or a discrete set of indices (in which case it can be instantiated using the syntax IndexVector(v), where v can represent either a real Vector or a list of non-negative integers, e.g. 1,7,32,5. Certain Vector operations can also return an IndexVector, e.g. find.

An IndexVector can be used to bulk index into another Vector or Matrix, either returning multiple or updating multiple elements at a time.

IndexVector supports the key operations of Vector and Matrix construction, which we consider next.

Vector construction

The Vector constructor operation allows the user to construct a new Vector from a set of continuous indices by supplying a restricted function that maps an index to a new value. We illustrate the use of vector constructor through a simple example:

```scala
// returns a DenseVector[Int] containing values (0,2,4,6,..)
val x = (0::100) { i => i*2 }
// returns a DenseVector[String] containing values ("1","2",...,)
val y = (0::100) { i => i.ToString }
```

Matrix construction

Matrix constructor is the 2-dimensional version of vector constructor. The function supplied to it must also be a restricted function. Matrix constructor can be invoked by supplying either scalars (every value
of the underlying Matrix), or vectors (every row of the underlying Matrix). We demonstrate each use case below:

```cpp
/*
 Matrix construction from scalars
 returns m x n result
 [0, 0, 0, 0, ...]
 [0, 1, 2, 3, ...]
 [0, 2, 4, 6, ...]
 .
 .
 */
 (0::m, 0::n) { (i,j) =>
   i*j
 }

/*
 Matrix construction from row vectors
 returns m x 4 result
 [0, 1, 2, 3]
 [0, 1, 2, 3]
 .
 .
 */
 (0::m, *) { i =>
   DenseVector(0,1,2,3)
 }
```

### 4.2.5 VectorView

A VectorView represents a view of an underlying Vector or Matrix. It is obtained by using a slice operator and provides a view of the underlying data as a contiguous Vector. Updates to a VectorView propagate back to the underlying data structure the view is based on.

### 4.2.6 Vertices

A DenseVector of Vertex instances. Supports a structured form of accessing neighbors during a parallel iteration, as described in section 5.1. Returned when accessing the vertices field of a Graph instance.

### 4.3 Matrix

OptiML's Matrix is a 2-dimensional matrix with an emphasis on row-major operations (since training sets in machine learning typically encode a training sample as a row). Like Vector, it is also polymorphic and can contain elements of any type, but provide arithmetic operations when used with arithmetic types. Below are the operations supported by Matrix:

---

4. Because OptiML's Matrix is row-major, some operations of Matrix are only available on rows for efficiency. This forces the user to transpose, rather than inadvertently suffer major performance losses.
There are also a number of static operations available for `Matrix`, listed below. These operations do not require a `Matrix` instance, and are used by calling `Matrix.<method>`.

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Matrix(Vector(a,b,c), Vector(d,e,f), ..)</code></td>
<td>return a new m x 3 dense matrix containing elements ((a,b,c),(d,e,f),..)</td>
</tr>
<tr>
<td><code>Matrix(m, n)</code></td>
<td>return a new m x n dense matrix</td>
</tr>
<tr>
<td><code>Matrix(v)</code></td>
<td>return a new dense matrix from v, a vector of vectors</td>
</tr>
<tr>
<td><code>Matrix.dense(m, n)</code></td>
<td>return a new m x n dense matrix</td>
</tr>
<tr>
<td><code>Matrix.sparse(m, n)</code></td>
<td>return a new m x n buildable sparse matrix</td>
</tr>
<tr>
<td><code>diag(w, vals)</code></td>
<td>return a new w x w diagonal matrix with values from vals on the diagonal</td>
</tr>
<tr>
<td><code>identity(w)</code></td>
<td>return a w x w identity matrix</td>
</tr>
<tr>
<td><code>zeros(m,n)</code></td>
<td>return a matrix of all double values 0.0</td>
</tr>
<tr>
<td><code>zerosf(m,n)</code></td>
<td>return a matrix of all float values 0.0</td>
</tr>
<tr>
<td><code>ones(m,n)</code></td>
<td>return a matrix of all double values 1.0</td>
</tr>
<tr>
<td><code>onesf(m,n)</code></td>
<td>return a matrix of all float values 1.0</td>
</tr>
<tr>
<td><code>rand(m,n)</code></td>
<td>return a matrix of random double values uniformly distributed from 0 to 1</td>
</tr>
<tr>
<td><code>randf(m,n)</code></td>
<td>return a matrix of random float values uniformly distributed from 0 to 1</td>
</tr>
<tr>
<td><code>randn(m,n)</code></td>
<td>return a matrix of random double values drawn from the standard normal distribution</td>
</tr>
<tr>
<td><code>randnf(m,n)</code></td>
<td>return a matrix of random float values drawn from the standard normal distribution</td>
</tr>
</tbody>
</table>

Table 10: Matrix static operations

Operations available on a `Matrix` instance are:

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>m(i,j)</code></td>
<td>return element at (i,j)</td>
</tr>
<tr>
<td><code>m(i,j) = x</code></td>
<td>set element at (i,j)</td>
</tr>
<tr>
<td><code>m(i)</code></td>
<td>return vector view of row i</td>
</tr>
<tr>
<td><code>m(i) = x</code></td>
<td>set row vector at i</td>
</tr>
<tr>
<td><code>getRow(i)</code></td>
<td>return vector view of row i</td>
</tr>
<tr>
<td><code>getCol(j)</code></td>
<td>return vector view of col j</td>
</tr>
<tr>
<td><code>vview(start,stride,len)</code></td>
<td>return a vector view of the matrix containing values every stride from start to length</td>
</tr>
<tr>
<td><code>slice(i0,i1,j0,j1)</code></td>
<td>submatrix from (i0-i1),(j0-j1)</td>
</tr>
<tr>
<td><code>sliceRows(s,e)</code></td>
<td>submatrix containing rows from s to e</td>
</tr>
<tr>
<td><code>numRows</code></td>
<td>return number of rows of m</td>
</tr>
<tr>
<td><code>numCols</code></td>
<td>return number of cols of m</td>
</tr>
<tr>
<td><code>size</code></td>
<td>return numRows*numCols</td>
</tr>
<tr>
<td><code>t</code></td>
<td>transpose</td>
</tr>
<tr>
<td><code>Clone</code></td>
<td>return an immutable deep-copy</td>
</tr>
<tr>
<td><code>mutable</code></td>
<td>return a mutable deep-copy</td>
</tr>
<tr>
<td><code>pprint</code></td>
<td>pretty-print m</td>
</tr>
<tr>
<td><code>replicate(i,j)</code></td>
<td>matrix with i,j tiles of m</td>
</tr>
<tr>
<td><code>&lt;&lt;(v)</code></td>
<td>return new matrix with appended row vector v</td>
</tr>
<tr>
<td><code>&lt;==(v)</code></td>
<td>append row vector v to m in place</td>
</tr>
<tr>
<td>Operation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>&lt;=(m2)</code></td>
<td>append matrix m2 to m in place; m2 must have the same number of cols as m</td>
</tr>
<tr>
<td>insertRow(n,x)</td>
<td>insert vector x at row n</td>
</tr>
<tr>
<td>insertAllRows(n,x)</td>
<td>insert matrix x at row n</td>
</tr>
<tr>
<td>insertCol(n,x)</td>
<td>insert vector x at col n</td>
</tr>
<tr>
<td>insertAllCols(n,x)</td>
<td>insert matrix x at col n</td>
</tr>
<tr>
<td>removeRow(n)</td>
<td>remove nth row</td>
</tr>
<tr>
<td>removeRows(n,len)</td>
<td>remove len rows starting at n</td>
</tr>
<tr>
<td>removeCol(n)</td>
<td>remove nth col</td>
</tr>
<tr>
<td>removeCols(n,len)</td>
<td>remove len cols starting at n</td>
</tr>
<tr>
<td>+(m2)</td>
<td>point-wise addition</td>
</tr>
<tr>
<td>-(m2)</td>
<td>point-wise substraction</td>
</tr>
<tr>
<td>*(m2)</td>
<td>point-wise multiplication</td>
</tr>
<tr>
<td>/ (m2)</td>
<td>point-wise division</td>
</tr>
<tr>
<td>+(m2)</td>
<td>matrix multiply</td>
</tr>
<tr>
<td>inv</td>
<td>matrix inverse</td>
</tr>
<tr>
<td>sum</td>
<td>sum of all elements in m</td>
</tr>
<tr>
<td>sumRow</td>
<td>return col vector containing sum of each row in m</td>
</tr>
<tr>
<td>sumCol</td>
<td>return row vector containing sum of each col in m</td>
</tr>
<tr>
<td>abs</td>
<td>point-wise absolute value</td>
</tr>
<tr>
<td>exp</td>
<td>point-wise exponentiation</td>
</tr>
<tr>
<td>sigmoid</td>
<td>return matrix of float values computed by the point-wise sigmoid function</td>
</tr>
<tr>
<td>sigmoidf</td>
<td>return matrix of double values computed by the point-wise sigmoid function</td>
</tr>
<tr>
<td>min</td>
<td>return min element</td>
</tr>
<tr>
<td>minRow</td>
<td>return row with minimum total value</td>
</tr>
<tr>
<td>max</td>
<td>return max element</td>
</tr>
<tr>
<td>maxRow</td>
<td>return row with maximum total value</td>
</tr>
<tr>
<td>&gt;(m2)</td>
<td>return boolean matrix representing point-wise &gt; comparison of m and m2</td>
</tr>
<tr>
<td>&lt;: (m2)</td>
<td>return boolean matrix representing point-wise &lt; comparison of m and m2</td>
</tr>
<tr>
<td>map(f)</td>
<td>transform each element in m</td>
</tr>
<tr>
<td>mmap(f)</td>
<td>transform m in place with a function f†</td>
</tr>
<tr>
<td>mapRows(f)</td>
<td>transform each row in m†</td>
</tr>
<tr>
<td>mapRowsToVector(f)</td>
<td>return dense vector computed by transforming each row in m to a scalar†</td>
</tr>
<tr>
<td>zip(m2,f)</td>
<td>return a new matrix as the result of applying f pair-wise to every element in m and m2 †</td>
</tr>
<tr>
<td>reduceRows(f)</td>
<td>flatten m with a reducing function†</td>
</tr>
<tr>
<td>filterRows(f)</td>
<td>filter m with a comparator function†</td>
</tr>
<tr>
<td>count(f)</td>
<td>return the number of elements in m matching the predicate f†</td>
</tr>
<tr>
<td>groupRowsBy(f)</td>
<td>return a dense vector of matrices, where each inner vector contains the matrix containing all rows that map to the same key using function f†</td>
</tr>
</tbody>
</table>

†Must be restricted functions (see Section 5)
4.3.1 DenseMatrix

The default Matrix in OptiML. A row-major, array-backed matrix. The most efficient format to use if the matrix is mostly non-zeros or is relatively small.

4.3.2 SparseMatrix

A matrix that physically stores only non-zero elements. Sparse matrices in OptiML have two phases, construction and operation, corresponding to different sparse representations. The constructor for SparseMatrix returns a new instance of SparseMatrixBuildable, which is implemented in a format that is efficient for inserting and updating new non-zero values (currently a coordinate list, i.e. COO format). SparseMatrixBuildables can be mutated, but they do not support any of the other normal Matrix operations (e.g. +). Calling finish on a SparseMatrixBuildable returns a SparseMatrix, which can then be used as a normal Matrix (except that it can no longer be mutated). Immutable sparse matrices are stored in a compressed sparse row (CSR) format, which is efficient for arithmetic and row slicing.

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SparseMatrix(m, n)</td>
<td>return a new m x n buildable sparse matrix equivalent to Matrix.sparse(m,n)</td>
</tr>
</tbody>
</table>

Table 12: SparseMatrix static operations

SparseMatrixBuildable instances support the mutable subset of Matrix along with the following additional methods:

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>finish</td>
<td>returns an immutable SparseMatrix version of m</td>
</tr>
</tbody>
</table>

Table 13: SparseMatrixBuildable class additional operations

SparseMatrix instances support the immutable subset of Matrix and the following additional methods:

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nnz</td>
<td>the number of non-zero values in m</td>
</tr>
</tbody>
</table>

Table 14: SparseMatrix class additional operations

4.3.3 Image

Image extends DenseMatrix to provide common operators over images:
### 4.3.4 TrainingSet

TrainingSet is a wrapper around Matrix. TrainingSet is immutable, and is the expected input for most algorithms in the OptiML standard library. SupervisedTrainingSet stores the data matrix along with a vector containing training labels, while UnsupervisedTrainingSet stores only the data matrix.

Although currently TrainingSet is mainly a convenience data structure in OptiML, the goal is to eventually have it provide more useful built-in machine learning primitives (e.g. ensembles), and to use the type information to inform device scheduling and data decomposition decisions.

A TrainingSet can be constructed using one of the two static methods below:

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SupervisedTrainingSet(m, v)</td>
<td>Construct a new training set with underlying data m and label vector v</td>
</tr>
</tbody>
</table>

Table 16: SupervisedTrainingSet static operations

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnsupervisedTrainingSet(m)</td>
<td>Construct a new training set with underlying data m</td>
</tr>
</tbody>
</table>

Table 17: UnsupervisedTrainingSet static operations

TrainingSet instances support the following methods:

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numSamples</td>
<td>the number of training samples (numRows in underlying matrix)</td>
</tr>
<tr>
<td>numFeatures</td>
<td>the number of features per sample (numCols in underlying matrix)</td>
</tr>
<tr>
<td>m(i,j)</td>
<td>return element at (i,j)</td>
</tr>
<tr>
<td>m(i)</td>
<td>return vector view of row i</td>
</tr>
<tr>
<td>data</td>
<td>return reference to underlying matrix</td>
</tr>
<tr>
<td>labels</td>
<td>return vector of labels (supervised only)</td>
</tr>
</tbody>
</table>

Table 18: TrainingSet class operations

### 4.4 Stream

Streams are meant to hold or compute a very large amount of data (more than can fit in memory). They are immutable, and only allow streaming access (i.e. next, prev) as well as bulk operators (e.g. foreach).
Stream only keeps a chunk of the actual underlying data in memory at a time, and transparently computes new chunks when required. Streams currently support the following operations:

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream(m, n, func)</td>
<td>construct a new m x n stream, where func computes the (ith,jth) element</td>
</tr>
</tbody>
</table>

Table 19: Stream static operations

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numRows</td>
<td>return number of rows of m</td>
</tr>
<tr>
<td>numCols</td>
<td>return number of cols of m</td>
</tr>
<tr>
<td>isPure</td>
<td>true if the stream initialization function is pure</td>
</tr>
</tbody>
</table>

Table 20: Stream class operations

4.5 Graph

Graph allows statistical inference problems on graphical models to be naturally expressed in OptiML. It provides a typical representation containing a set of Vertices and Edges. Vertices and Edges can be iterated over with built-in synchronization, as described in section 5. Graph is designed for message passing; Vertices can store data, and Edges can store both incoming and outgoing data.

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph()</td>
<td>construct a new graph instance</td>
</tr>
</tbody>
</table>

Table 21: Graph static operations

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertices</td>
<td>vector containing vertices of g</td>
</tr>
<tr>
<td>edges</td>
<td>vector containing edges of g</td>
</tr>
<tr>
<td>neighborsOf(v)</td>
<td>vector containing neighbors of vertex v</td>
</tr>
<tr>
<td>edgesOf(v)</td>
<td>vector containing edges of vertex v</td>
</tr>
<tr>
<td>addVertex(v)</td>
<td>add a vertex v to g</td>
</tr>
<tr>
<td>addEdge(e)</td>
<td>add an edge e to g</td>
</tr>
<tr>
<td>freeze</td>
<td>finalize the graph structure</td>
</tr>
</tbody>
</table>

Table 22: Graph class operations

4.5.1 Vertex

Vertex is represents a node in a Graph. A Vertex can store data of any OptiML type (or user-defined struct containing OptiML types).

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex(g, d)</td>
<td>construct a new Vertex on graph g storing data d</td>
</tr>
</tbody>
</table>

Table 23: Vertex static operations
Table 24: Vertex class operations

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>edges</td>
<td>return the set of edges connected to this vertex</td>
</tr>
<tr>
<td>neighbors</td>
<td>return the set of neighboring vertices</td>
</tr>
<tr>
<td>data</td>
<td>return the data stored at this vertex</td>
</tr>
<tr>
<td>graph</td>
<td>return the graph associated with this vertex</td>
</tr>
</tbody>
</table>

4.5.2 Edge

Edge represents an edge in a Graph. An Edge can store both ingoing and outgoing data of any OptiML type (or user-defined struct containing OptiML types).

Table 25: Edge static operations

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge(g, dIn, dOut, a, b)</td>
<td>construct a new Edge on graph g, from vertex a to vertex b with dIn stored as field inData and dOut stored as field outData</td>
</tr>
</tbody>
</table>

Table 26: Edge class operations

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inData</td>
<td>return the data stored as an incoming message</td>
</tr>
<tr>
<td>outData</td>
<td>return the data stored as an outgoing message</td>
</tr>
<tr>
<td>v1</td>
<td>return the first vertex connected to this edge</td>
</tr>
<tr>
<td>v2</td>
<td>return the second vertex connected to this edge</td>
</tr>
<tr>
<td>graph</td>
<td>return the graph associated with this edge</td>
</tr>
</tbody>
</table>

4.6 User-defined types

OptiML allows struct-like data structures to be defined by users. These data structures must only contain fields (cannot contain any methods). A user-defined type can be declared and used by defining a new Record:

```scala
val MyType = new Record {
    val x: Int
    val y: String
    val z: DenseVector[Int]
}

type RT = Record{
    val x: Int;
    val y: String;
    val z: DenseVector[Int]
}

val v = DenseVector[RT](100, true)
```
5 Control structures

5.1 Iterating with 'for'

It is possible to iterate over the elements of an OptiML data structure using the the special for iterator. Each iterator accepts any restricted function, but in certain cases, the restrictions are relaxed to allow structured communication. The access patterns are described below:

<table>
<thead>
<tr>
<th>operation</th>
<th>access pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (e &lt;- Vector)</td>
<td>cannot write to any other elements in Vector</td>
</tr>
<tr>
<td>for (v &lt;- Vertices)</td>
<td>can only write to neighboring vertices, e.g. v.neighbors</td>
</tr>
<tr>
<td>for (e &lt;- Matrix)</td>
<td>cannot write to any other elements in Matrix</td>
</tr>
<tr>
<td>for (r &lt;- Matrix.rows)</td>
<td>cannot write to any other rows in Matrix</td>
</tr>
<tr>
<td>for (p &lt;- Image)</td>
<td>can only write to pixels in a local window, e.g. p.window</td>
</tr>
<tr>
<td>for (r &lt;- Stream.rows)</td>
<td>cannot access any other rows in Stream</td>
</tr>
</tbody>
</table>

Table 27: For operations

Note that each of these operations are syntactic sugar for the corresponding foreach methods defined on each data type.

5.2 untilconverged

untilconverged is an iterative control structure that iterates until reaching a convergence criterion. Each iteration of untilconverged produces a value, and the loop converges when the difference between values in consecutive iterations falls below a supplied threshold parameter, or a maximum number of iterations is reached.

untilconverged has the following signature:

\[
\text{untilconverged} \left[ A \right] (x: \text{Rep}[A], \text{thresh}: \text{Rep}[\text{Double}], \text{max}_\text{iter}: \text{Rep}[\text{Int}] = \text{unit}(1000), \text{clone}_\text{prev}_\text{val}: \text{Rep}[\text{Boolean}] = \text{unit}(\text{false})) \\
(\text{block}: \text{Rep}[A] \Rightarrow \text{Rep}[A]) \\
(\text{implicit diff:} \ (\text{Rep}[A], \text{Rep}[A]) \Rightarrow \text{Rep}[\text{Double}])
\]

By overriding default parameters, you can configure untilconverged to have a different maximum number of iterations, to clone the result of the previous iteration before passing it to the next (so mutations in this iteration don’t effect previous values), or to use a custom difference function (diff) when comparing the result of this iteration to the previous iteration.

An example of using untilconverged is shown below:

```scala
val x = Matrix.ones(100,100) * 1000
// default matrix difference is absolute difference
val x2 = untilconverged(x, .01) {
    m =>
    m / 100
}
```
5.3 gradient

gradient is a specialized version of untilconverged that implements the gradient descent algorithm for exponential family models. Specifically, it provides batch and stochastic variants (the default is batch).

The batch variant has the following signature:

\[
\text{batch}(x: \text{Rep[TrainingSet[Double]]}, \alpha: \text{Rep[Double]}, \text{thresh}: \text{Rep[Double]}, \text{maxIter: Rep[Int]})
\]

\[
(\text{hyp: Rep[Vector[Double]]} => \text{Rep[Double]}): \text{Rep[Double]}
\]

where alpha is the learning rate, thresh is the convergence threshold, maxIter is the maximum number of iterations, and hyp is the hypothesis function that maps a training sample to a scalar value. batch implements the following pseudocode:

\[
\text{block}() \text{ updates each } j\text{th parameter from the sum of all } i\text{th training samples}
\]

\[
\text{while not converged}
\]

\[
\text{for } j \text{ from 0 until } n
\]

\[
j\_update = \text{sum}((y(i) - h(x(i))^s(x(j,i)
\]

\[
\text{update}(j\_update)
\]

The stochastic variant has the same signature as batch, but implements the following pseudocode:

\[
\text{block}() \text{ updates every } j\text{th parameter for every } i\text{th training sample}
\]

\[
\text{while not converged}
\]

\[
\text{for } i \text{ from 0 until } m
\]

\[
\text{for } j \text{ from 0 until } n
\]

\[
\text{update}(i,j)
\]

6 Built-in functions

OptiML supplies a number of built-in functions that are useful for statistics and machine learning.

In addition to the functions listed below, the following standard mathematical operators are available on scalars, sequences of numbers, vectors, or matrices:

- abs, min, max, mean, exp, log

The following operators are available on scalars, sequences of numbers, or vectors only:

- median

The following operators are available on scalar Double values only:

- sqrt, ceil, floor, pow, sin, cos, acos, atan, atan2

The following standard mathematical constants are also available:

- Pi, E, INF, nINF
6.1 sum

sum expresses generic summations over an indexed range, i.e. \( \sum_{i=0}^{N-1} f(i) \) where \( f(i) \) is a user-defined anonymous function\( ^\dagger \).

Below is an example of computing a simple scalar sum:

```scala
val x = sum(0,100) { i => exp(i) }
```

Sum can also be used to accumulate vectors or matrices. A specialized version, `sumRows`, should be used when summing over vector views (e.g. from matrix rows); this version adds the values from the underlying matrix directly into the accumulator without creating a copy of each row. Below is an example of using `sumRows`:

```scala
val m = Matrix.ones(10,10)
val x = sumRows(0,10) { i => m(i) }
```

There are also conditional versions `sumIf` and `sumRowsIf`. These variants allow specifying a predicate before the summation and will only add the computed value if the predicate is true for that index:

```scala
// sum(exp(0),exp(2),exp(4)...)  
val x = sumIf(0,100) (i => i % 2 == 0) { i => exp(i) }

val m = Matrix.rand(10,10)  
// only sums rows whose minimum value is at least .1  
val y = sumRowsIf(0,10)(i => m(i).min > .1) { i => m(i) }
```

\( ^\dagger \)Must be a restricted function (see Section 3)

6.2 aggregate

aggregate is similar to `sum`, but is used to concatenate values instead of add them. The values computed in an aggregate are appended to a `DenseVector`, which is returned as the result. The primary form of aggregate is a 2-dimensional form, where the aggregation occurs over a `rowIndexVector` and a `colIndexVector`\( ^\dagger \):

```scala
// result is [ 30 40 50 33 44 55 36 48 60 ]  
val x = aggregate(10::13, 3::6) { (i,j) => i*j }
```

Like `sumIf`, there is also the conditional version `aggregateIf`:

```scala
// result is [ 0 0 0 2 3 6 ]  
val x = aggregateIf(0::3, 0::4) { (i,j) => i < j } { (i,j) => i*j }
```

`aggregateIf` can also be used in a 1-dimensional form, with a scalar start and end index:

```scala
// result is [ 10 12 14 16 18 ]  
val x = aggregateIf(0, 10) { i => i > 4 } { i => i*2 }
```

Note that there is no unconditional, 1-dimensional form of `aggregate`, since that is identical to a `vector constructor` (Sec. 4.2.4).  

\( ^\dagger \)Must be a restricted function (see Section 3)
6.3 dist

Dist computes the distance between two vectors or matrices according a distance metric, of which OptiML has three pre-defined variants (ABS, SQUARE, EUC). The distances computed for these metrics are (assuming two vectors, or two matrices, \( p \) and \( q \)):

- **abs**: \( \sum_{i=0}^{N-1} |p_i - q_i| \)
- **square**: \( \sum_{i=0}^{N-1} (p_i - q_i)^2 \)
- **euclidean**: \( \sqrt{\sum_{i=0}^{N-1} (p_i - q_i)^2} \)

Distances are calculated by simply passing the vector or matrix arguments to the `dist` function, with an optional metric parameter (the default metric is ABS):

```plaintext
val v1 = Vector.rand(100)
val v2 = Vector.rand(100)
val d = dist(v1,v2)
val d2 = dist(v1,v2,EUC)
```

dist can also be used to compute the magnitude difference of two doubles, i.e. |\( d_1 - d_2 \)|:

```plaintext
val d = dist(130513.5, -3356.35)
```

6.4 det

Det computes the determinant of the input matrix \( m \):

```plaintext
val m = Matrix.rand(10,10)
val d = det(m)
```

6.5 sample

Sample takes a pseudorandom sample of a vector or matrix, returning a new vector or matrix with elements or rows/cols drawn from the original. The first argument to sample is the number of samples to take, and the second argument (when sampling a matrix) is a boolean `sampleRows`, which if true draws rows from the original matrix and if false draws columns.

```plaintext
val v = Vector.rand(100) // length 100
val sampV = sample(v, 10) // length 10, elements drawn uniformly from v
val m = Matrix.rand(10,10)
val sampM = sample(m, 5) // 5 x 10, five rows drawn uniformly from m
val sampMC = sample(m, 5, false) // 10 x 5, five cols drawn uniformly from m
```
6.6 nearestNeighborIndex

This function computes the nearest neighbor of a row \( r \) in a matrix \( m \) to all the other rows in \( m \). It accepts an optional boolean parameter, allowSame, which specifies whether or not a row that matches exactly should be considered a nearest neighbor (if true, then it will match). nearestNeighborIndex uses the dist function (with metric \( \text{ABS} \)) to compute the nearest neighbor.

```scala
val m = Matrix.rand(100,10)
// find the index of the row in m that has values closest to row 10

// may return the index of an identical row
val nearestNeighbor = nearestNeighborIndex(10, m)

// will not return the index of an identical row
val nearestNeighbor2 = nearestNeighborIndex(10, m, false)
```

6.7 random[T]

The random[T] function is used to generate a new pseudorandom value of a particular type. The type arguments that it accepts are Double, Float, Int, and Boolean. The specification for each variant follows:

- random[Double]: uniformly distributed double between 0.0 and 1.0
- random[Float]: uniformly distributed float between 0.0 and 1.0
- random[Int]: uniformly distributed int between 0 and \( 2^{31} - 1 \)
- random[Boolean]: either true or false with (approximately) equal probability

There is a final variant, randomGaussian, that returns a normally distributed double with mean 0.0 and standard deviation 1.0.

OptiML seeds all random number generators at the beginning of each run to a known initial value in order to compare execution time across runs. The function reseed can be used to reset the seed to this value at any time.

6.8 Profiling

OptiML has two methods for application-level profiling: the tic/toc functions and the time function.

In the same way as MATLAB, tic and toc calls must be paired and OptiML will print out the elapsed execution time (in seconds) between them. You can also have nested tic/toc calls by naming them:

```scala
tic()
while(..) {
  tic("while")
  // do some stuff
  toc("while")
}
toc()
```

\(^5\)No seed function exists yet, but this will be addressed in a future release.
You can also specify dependencies that must run before a tic or a toc to prevent code motion from moving them around while optimizing:

```scala
val z = lotsOfWork()
// tic("x") won't run until z completes
tic("x", z)
val x = lotsOfMoreWork()
// toc("x") won't run until x completes
toc("x", x)
```

OptiML also has a time() function similar to Python's time(). You can also supply it dependencies, and then you can take the difference of two time() calls to return elapsed time in seconds:

```scala
val st = time()
val x = lotsOfWork()
val en = time(x)
println("elapsed: " + en-st)
```

### 6.9 I/O

OptiML currently supports reading and writing files as a tab-delimited set of values, where matrices are represented by multiple lines (this is the same default format as MATLAB).

The functions to read files are:

- `readVector(filename)`
- `readMatrix(filename)`
- `readGrayscaleImage(filename)`
- `readARFF(filename, schema)`

`readVector` and `readMatrix` accept optional `schema` and `delim` parameters. For `readVector`, `schema` is a function that specifies how to convert a delimited line (represented as a `Vector[String]`) to a single vector value. For `readMatrix`, `schema` is a function that specifies how to convert a delimited value (represented as a `String`) to a single matrix value. For both versions, `delim` is a `String` specifying a different delimiter than the default tab.

The generic versions of `readVector` and `readMatrix` are used in the following way:

```scala
// ignores everything in each line after the first ';'
val v = readVector[Double]("myvector.dat", line => line(0).toDouble, ";")

// uses a semicolon delimiter instead of a tab delimiter, and drops everything
// after the decimal point
val m = readMatrix[Double]("mymatrix.dat", s => (s.split("."))(0).toInt, ";")

// uses a semicolon delimiter, but with the default schema (s.toDouble)
val m2 = readMatrix[Double]("mymatrix.dat", ";")
```

`readARFF` is used to read WEKA ARFF files, as described at the following URL: [http://www.cs.waikato.ac.nz/ml/weka/arff.html](http://www.cs.waikato.ac.nz/ml/weka/arff.html)
Like readVector, the schema argument to readARFF specifies how to construct a single vector value from a delimited line. The following snippet shows how to use OptiML to read an example ARFF file:

```scala
def mySchema(v: Rep[DenseVector[String]]) =
  new Record {
    val sepalLength = v(0).toDouble
    val sepalWidth = v(1).toDouble
    val petalLength = v(2).toDouble
    val petalWidth = v(3).toDouble
    val cls = v(4)
  }

val in = readARFF(args(0), mySchema)
println("First row sepalLength is: " + in(0).sepalLength))
```

The functions to write files are:

- `writeVector(v, filename)`
- `writeMatrix(m, filename)`
- `writeImagePgm(img, filename)`

`writeVector` and `writeMatrix` write files as a tab-delimited set of values (the same as the default input format). `writeImagePgm` writes a grayscale image out as a PGM (Portable Gray Map) image.

## 7 Optimizations

OptiML performs several optimizations, many of which are inherited from the Delite framework [1]. The optimizations can be split into static and dynamic depending on whether they are applied during staging (static) or during execution (dynamic).

### 7.1 Static optimizations

OptiML implements the following well-known compiler optimizations:

**Common subexpression elimination:** OptiML tracks DSL operations that are pure versus ones that may have side-effect; if a pure operation is used multiple times with the same input (e.g. \( x + y \) is calculated twice with the same \( x \) and \( y \)), OptiML will reduce these to the same symbol and only calculate it once.

**Dead code elimination:** if the result of an operation is not used anywhere as part of the final output of the program, and if it is pure, OptiML will eliminate it entirely from the generated code.

**Loop hoisting:** any operations inside a loop that do not need to be recalculated every iteration are pulled out of the loop; similarly, operations are pushed into control statements as far as possible so that they are only run if necessary.

Although these are traditional optimizations, it is important to note that in the Delite framework they occur on symbols that represent coarse-grained domain-specific operations (such as `MatrixMultiply`), rather than generic language statements as in a typical general-purpose compiler.
**Fusion:** probably the most impactful optimization provided by OptiML is *op fusion*, wherein data parallel operations that have a producer/consumer or sibling relationship are fused together, eliminating temporary allocations and extra memory accesses. These optimizations are especially important for linear algebra on large vectors and matrices, as it avoids constructing expensive intermediate objects when doing sequences of arithmetic such as \( v_1 + v_2 \times s/100 \).

As a practical example, consider the following line from the SMO algorithm [6] for SVM (*:* is a dot product) [9]:

```plaintext
val eta = (X(i)::*:*X(j)::2) - (X(i)::*:*X(i)) - (X(j)::*:*X(j))
```

Here, OptiML automatically fuses all of the dot product calculations into a single loop instead of 4 (3 for each dot product plus 1 for the scalar multiplication). For the entire SMO algorithm, op fusing reduces 35 loops to 11. More importantly, fusing an operation can eliminate allocations of intermediate data structures. The *:*:* operator can be implemented as \((X(i) \times X(j)).\text{sum}\) and op fusing will ensure that no intermediate vector will be allocated to hold the result of \(X(i) \times X(j)\).

**Linear algebra simplification:** finally, OptiML also uses pattern matching to apply well-known linear algebra simplifications to expressions in order to reduce overall computation. For example, expressions like

```plaintext
val x = Vector.rand(100) + Vector.zeros(100)
```

will be rewritten to eliminate the addition.

### 7.2 Dynamic optimizations

OptiML provides constructs that dynamically take advantage of the characteristics of machine learning algorithms in order to improve performance [9].

**Best-effort computing:** because many ML algorithms are iterative and probabilistic, they are often robust to minor variations in computation [4]. OptiML allows users to trade-off accuracy, if they choose, for better performance, by using best effort data structures. These data structures drop computations according to a policy, which can improve single-threaded execution time and reduce sequential bottlenecks, improving parallel scalability.

**Relaxed dependencies:** for the same reasons as above, it is sometimes useful to allow ML algorithms to intentionally race, which again can improve parallel performance at the expense of strict consistency for some operations. OptiML provides a version of the `until-converged` construct that allows some number of iterations to be run in parallel. Recent work [10] has shown the potential for this optimization.

---

6These features were available in previous versions of OptiML but are not currently part of `optiml-beta`. They are scheduled to be added again in a future release.
8  Advanced Topics

8.1  Type system

OptiML inherits its type system from Scala, and any Scala type annotations may be used. However, to use these features, the user must keep in mind that all OptiML types are wrapped in the \textit{Rep} type. For example, one could specify two illustrative type-bounds on methods as follows:

\begin{verbatim}
def foo[A <: Double](x: Rep[Vector[A]]) { ... }
def bar[B <: Rep[Vector[Double]]](x: B) { ... }
\end{verbatim}

Of course, using advanced type signatures is not required. In most cases, Scala’s type inference allows OptiML result types to be inferred with no type declaration necessary.

A detailed description of Scala’s type system can be found at \cite{5}.

8.2  Pass by value

OptiML also follows the Scala/Java pass by value semantics, where objects are \textit{passed by the value of their reference}, i.e., an immutable reference to the object. Therefore, accessing an object element in OptiML data structures returns a the reference to the object itself (not a copy); in a \texttt{Vector[Foo]}, returning \texttt{v(0)} will return a reference to the Foo object at element 0. Similarly, accessing row \texttt{i} of a Matrix using \texttt{m(i)} returns a reference to a new \texttt{VectorView} representing the elements at that position. Accessing a primitive object, e.g. \texttt{v(0)} in a \texttt{Vector[Double]} will return the value of the double of at \texttt{v(0)}.

8.3  Mutable/Immutable operators

Almost all OptiML operations on a data structure return an entirely new data structure. For example, \texttt{v + 5} if \texttt{v} is a \texttt{Vector} will not modify \texttt{v}, but will return an entirely new \texttt{Vector}. There are a small number of operations that do make in-place modifications which are noted in section \cite{4}

Although a naive implementation of these semantics could result in a large number of unnecessary temporary allocations, the fusion optimization described in section \cite{7} aggressively eliminates these when possible.

8.4  Large scale program organization

All OptiML code must be executed within the dynamic scope of the \texttt{main} method in order to be staged. However, you can still organize your code into separate methods and traits (and different traits can be put in different source files). A 'trait' is a type of Scala class that supports mix-in inheritance using the \texttt{with} keyword. Note that global variables outside of the main method are not allowed, since this isn’t within the dynamic scope. Therefore, methods either require all of their parameters to be passed in explicitly, or must be declared as sub-methods within \texttt{main}.

The following example demonstrates composing an application out of two traits:
object ExampleRunner extends OptiMLApplicationRunner with Example
trait Example extends OptiMLApplication with ExampleWork {
  def main() = {
    // code can be organized into different methods and traits
    // these methods get inlined during staging
    val v = Vector.rand(1000)
    doWork(v) // defined in ExampleWork
  }
}

// could be in a different file
trait ExampleWork extends OptiMLApplication {
  def doWork(v: Rep[DenseVector[Double]]) = {
    println("v length is: " + v.length)
  }
}

8.5 Interfaces

All OptiML methods are statically dispatched in order to be easier to retarget to different programming models (e.g. CUDA, OpenCL). However, this restriction does complicate typical object-oriented generic programming which relies heavily on dynamic dispatch.

In order to still support the common cases of generic programming, OptiML uses a combination of type classes and interfaces. A type class provides an implicit dictionary, called the type class instance, to generic functions. This dictionary describes how a particular object implements a particular interface. One example of this in OptiML is the Arith type class (see Section 8.6).

An OptiML Interface wraps the type class within an object instance in a way that reflects the inheritance hierarchy (similar to traditional OO-style programming). The type Interface[T] represents a uniform interface for generic types (e.g. Vector[T] that can have different internal representations (e.g. SparseVector[T] and DenseVector[T]). Any specialized vector instance can be converted to an interface, but an interface cannot be converted back to a specialized instance without an explicit type class (since in general, the specific type of a particular interface instance is not known).

Interfaces can be used as follows:

// you can use the Interface[T] type to abstract over the subtype of
// Vector being passed in, e.g. if you don’t care if it’s dense or sparse
def doWork[T:Manifest](v: Interface[Vector[T]]) = {
  println("v is a row vector: " + v.isRow)
}

val v1 = Vector.dense(100)
doWork(v1)

val v2 = Vector.sparse(100)
doWork(v2)

val m = Matrix.ones(10,10)
val vview: Rep[DenseVectorView[Double]] = m(5)
doWork(vview)
8.6 Adding functionality to user-defined types

OptiML's data structures (e.g. vector) are generic and can operate on arbitrary types. However, certain operations like math are restricted to types that support arithmetic operations using type classes. You can extend a new type to support arithmetic by adding a new instance of the Arith type class. For example, if we first define a new type MyStruct:

```scala
// type alias is for convenience
type MyStruct = Record{val data: Int; val name: String}

// method to construct a new instance of MyStruct, also for convenience
def newMyStruct(_data: Rep[Int], _name: Rep[String]) =
  // a user-defined struct instance is declared as a new Record
  new Record {
    val data = _data
    val name = _name
  }
```

We can then add arithmetic capability to MyStruct:

```scala
// we can even use math operators by defining how arithmetic works with MyStruct
    newMyStruct(a.data+b.data,a.name)
    newMyStruct(a.data+b.data,a.name + " pl "+b.name)
    newMyStruct(a.data-b.data,a.name + " mi "+b.name)
    newMyStruct(a.data*b.data,a.name + " ti "+b.name)
    newMyStruct(a.data/b.data,a.name + " di "+b.name)
  def abs(a: Rep[MyStruct])(implicit ctx: SourceContext) =
    newMyStruct(arith_abs(a.data),"abs " +a.name)
  def exp(a: Rep[MyStruct])(implicit ctx: SourceContext) =
    newMyStruct(arith_exp(a.data).AsInstanceOf[Int],"exp " +a.name)
  def empty(implicit ctx: SourceContext) =
    newMyStruct(0,"empty")
  def zero(a: Rep[MyStruct])(implicit ctx: SourceContext) =
    newMyStruct(0,"zero")
}
```

And then use it as follows:

```scala
val v1 = (0::100) { i => newMyStruct(i, "struct1 " + i) }
val v2 = (0::100) { i => newMyStruct(42, "struct2 " + i) }
val result = v1+v2
println("result(10) with name " + result(10).name + " has data " + result(10).data)
```

In addition to Arith, OptiML supports Cloneable, HasMinMax, HasZero, and CanSum type classes, which are required for different methods.
9 Library

The OptiML standard library is small, but growing. The goal is to eventually contain a number of off-the-shelf implementations of common machine learning algorithms, written in OptiML, that are useful for someone writing applications that need to leverage machine learning algorithms (as opposed to designing their own algorithm).

Currently available library functions are linear regression and k-means clustering. These functions can be found in the ppl.dsl.optiml.library package, and example applications using them can be found in the ppl.apps.ml package.

References


