OptiML Language Specification 0.2

Arvind K. Sujeeth
Stanford University
asujeeth@stanford.edu

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 catagories. 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:

```
  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.[1] 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:

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

2.5 Methods

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

```
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.[2]

Here is an example of using wrapped method parameters:

```
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[3]:

operation	description
IsInstanceOf	dynamic type test
AsInstanceOf	type cast
ToString	string representation

Table 1: Generic operations

4.1 Primitives

4.1.1 Numerics

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

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

operation	description
+	addition
-	subtraction
*	multiplication
/	division
abs	absolute value
exp	base e exponentiation

Table 2: Numeric operations

4.1.2 Boolean

The `Boolean` type supports:

operation	description
!	unary negation
&&	logical and
||	logical or

Table 3: Boolean operations

4.1.3 String

The `String` type supports:

operation	description
+	string concatentation
trim	returns a new string with whitespace removed
split	returns an array of strings
startsWith(prefix)	tests if the string starts with the given prefix
toDouble	converts the string to a double
toFloat	converts the string to a float
toInt	converts the string to an int

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:

```
// 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
```

```
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 `Unit`, i.e. `void`). Currently only functions up to 5 arguments are allowed. For example,

```
val newVector: Rep[Int] => Rep[DenseVector[Int]] = n => DenseVector[Int](n)
val vecFive = newVector(5)
```

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 `Vector` from `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 `Vector.<method>`.

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

Table 5: Vector static operations

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

operation	description
toBoolean	converts v to boolean if an element-wise conversion exists
toDouble	converts v to double if an element-wise conversion exists
toFloat	converts v to float if an element-wise conversion exists
toInt	converts v to int if an element-wise conversion exists
v(i)	returns elements with indices in i
v(i) = x	set elements in i to value x
length	returns the length of v
isRow	true if v is a row vector

isEmpty	true if v is empty
first	returns the first element of v
last	returns the last element of v
indices	IndexVector from 0 to length
drop(c)	returns new vector without the first c elements of v
take(c)	returns new vector with only the first c elements of v
slice(s,e)	returns new vector with elements from s to e
contains(y)	true if y is in v
distinct	returns new vector with only unique elements in v
t	transpose
mt	in-place transpose (modifies v)
Clone	return an immutable deep-copy
mutable	return a mutable deep-copy
pprint	pretty-print v
replicate(i,j)	matrix with i,j tiles of v
mkString(sep)	return a string formed by joining all the elements in v with the 'sep' string
<<(e)	return new vector with appended element e
<<=(e)	append element e to v in place
<<(v2)	return new vector with appended vector v2
<<=(v2)	append vector v2 to v in place
copyFrom(p,v2)	overwrite elements of v at index p with elements from v2
insert(p,e)	insert element e into v at index p, in place
insertAll(p,v2)	insert all elements from v2 into v at index p, in place
remove(p)	remove the element at index p from v, in place
removeAll(p, len)	remove len elements starting at index p from v, in place
trim	trim the backing store of v to its logical length
clear	removes all elements from v, in place
+(v2)	point-wise addition
+=(v2)	point-wise addition in place
+=(e)	point-wise addition of e to every element of v in place
∗(v2)	point-wise multiplication
∗=(v2)	point-wise multiplication in place
∗=(e)	point-wise multiplication of e to every element of v in place
-(v2)	point-wise subtraction
-=(v2)	point-wise subtraction in place
-=(e)	point-wise subtraction of e to every element of v in place
/(v2)	point-wise division
/=(v2)	point-wise division in place
/=(e)	point-wise division of e to every element of v in place
∗(m)	vector-matrix product
∗∗(v2)	outer product
∗:∗(v2)	dot product
sum	sum of all elements in v
abs	point-wise absolute value
exp	point-wise exponentiation
sort	sorted vector v

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:

operation	description
DenseVector(len, isRow)	construct a new empty dense vector of initial size len and orientation based on boolean isRow equivalent to Vector.dense(len,isRow)
flatten(vectors)	concatenates a vector of vectors into a single vector

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:

operation	description
SparseVector(len, isRow)	construct a new empty sparse vector of initial size len and orientation based on boolean isRow equivalent to Vector.sparse(len,isRow)

Table 8: SparseVector static operations

`SparseVector` instances also support additional methods:

operation	description
nnz	the number of non-zero values in v
mapNZ(f)	transform v by applying a function f to each non-zero element[†] the resulting vector is also sparse and has the same size (logical and nnz) as v

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:

```
// 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.toStringL }
```

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:

```
/*
   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]:

[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 inadvertantly 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>`.

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

Table 10: Matrix static operations

Operations available on a `Matrix` instance are:

operation	description
toBoolean	converts m to boolean if an element-wise conversion exists
toDouble	converts m to double if an element-wise conversion exists
toFloat	converts m to float if an element-wise conversion exists
toInt	converts m to int if an element-wise conversion exists
m(i,j)	return element at (i,j)
m(i,j) = x	set element at (i,j)
m(i)	return vector view of row i
m(i) = x	set row vector at i
getRow(i)	return vector view of row i
getCol(j)	return vector view of col j
vview(start,stride,len)	return a vector view of the matrix containing values every stride from start to length
slice(i0,i1,j0,j1)	submatrix from (i0-i1),(j0-j1)
sliceRows(s,e)	submatrix containing rows from s to e
numRows	return number of rows of m
numCols	return number of cols of m
size	return numRows*numCols
t	transpose
Clone	return an immutable deep-copy
mutable	return a mutable deep-copy
pprint	pretty-print m
replicate(i,j)	matrix with i,j tiles of m
<<(v)	return new matrix with appended row vector v
<<=(v)	append row vector v to m in place

<<=(m2)	append matrix m2 to m in place; m2 must have the same number of cols as m
insertRow(n,x)	insert vector x at row n
insertAllRows(n,x)	insert matrix x at row n
insertCol(n,x)	insert vector x at col n
insertAllCols(n,x)	insert matrix x at col n
removeRow(n)	remove nth row
removeRows(n,len)	remove len rows starting at n
removeCol(n)	remove nth col
removeCols(n,len)	remove len cols starting at n
+(m2)	point-wise addition
-(m2)	point-wise substraction
∗:∗(m2)	point-wise multiplication
/(m2)	point-wise division
∗(m2)	matrix multiply
inv	matrix inverse
sum	sum of all elements in m
sumRow	return col vector containing sum of each row in m
sumCol	return row vector containing sum of each col in m
abs	point-wise absolute value
exp	point-wise exponentiation
sigmoid	return matrix of float values computed by the point-wise sigmoid function
sigmoidf	return matrix of double values computed by the point-wise sigmoid function
min	return min element
minRow	return row with minimum total value
max	return max element
maxRow	return row with maximum total value
:>(m2)	return boolean matrix representing point-wise > comparison of m and m2
:<(m2)	return boolean matrix representing point-wise < comparison of m and m2
map(f)	transform each element in m†
mmap(f)	transform m in place with a function f†
mapRows(f)	transform each row in m†
mapRowsToVector(f)	return dense vector computed by transforming each row in m to a scalar†
zip(m2,f)	return a new matrix as the result of applying f pair-wise to every element in m and m2†
reduceRows(f)	flatten m with a reducing function†
filterRows(f)	filter m with a comparator function†
count(f)	return the number of elements in m matching the predicate f†
groupRowsBy(f)	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†

Table 11: Matrix class operations

†Must be restricted functions (see Section 3)

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.

operation	description
SparseMatrix(m, n)	return a new m x n buildable sparse matrix equivalent to Matrix.sparse(m,n)

Table 12: SparseMatrix static operations

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

operation	description
finish	returns an immutable `SparseMatrix` version of m

Table 13: SparseMatrixBuildable class additional operations

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

operation	description
nnz	the number of non-zero values in m

Table 14: SparseMatrix class additional operations

4.3.3 Image

`Image` extends `DenseMatrix` to provide common operators over images:

operation	description
downsample	return a new image by converting blocks of the original image to a single value
windowedFilter	return a new image computed by running a rectangular windowed filter over the original image
convolve(kernel)	computes a new image where each pixel is the sum of the pointwise multiplication of the given kernel and a window of the original image

Table 15: Image operations

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:

operation	description
SupervisedTrainingSet(m, v)	Construct a new training set with underlying data m and label vector v

Table 16: SupervisedTrainingSet static operations

operation	description
UnsupervisedTrainingSet(m)	Construct a new training set with underlying data m

Table 17: UnsupervisedTrainingSet static operations

`TrainingSet` instances support the following methods:

operation	description
numSamples	the number of training samples (numRows in underlying matrix)
numFeatures	the number of features per sample (numCols in underlying matrix)
m(i,j)	return element at (i,j)
m(i)	return vector view of row i
data	return reference to underlying matrix
labels	return vector of labels (supervised only)

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:

operation	description
Stream(m, n, func)	construct a new m x n stream, where func computes the (ith,jth) element

Table 19: Stream static operations

operation	description
numRows	return number of rows of m
numCols	return number of cols of m
isPure	true if the stream initialization function is pure

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.

operation	description
Graph()	construct a new graph instance

Table 21: Graph static operations

operation	description
vertices	vector containing vertices of g
edges	vector containing edges of g
neighborsOf(v)	vector containing neighbors of vertex v
edgesOf(v)	vector containing edges of vertex v
addVertex(v)	add a vertex v to g
addEdge(e)	add an edge e to g
freeze	finalize the graph structure

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).

operation	description
Vertex(g, d)	construct a new Vertex on graph g storing data d

Table 23: Vertex static operations

operation	description
edges	return the set of edges connected to this vertex
neighbors	return the set of neighboring vertices
data	return the data stored at this vertex
graph	return the graph associated with this vertex

Table 24: Vertex class operations

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).

operation	description
Edge(g, dIn, dOut, a, b)	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

Table 25: Edge static operations

operation	description
inData	return the data stored as an incoming message
outData	return the data stored as an outgoing message
v1	return the first vertex connected to this edge
v2	return the second vertex connected to this edge
graph	return the graph associated with this edge

Table 26: Edge class operations

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`:

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

  val v = DenseVector[RT](100)
}
```

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:

operation	access pattern
for (e <- Vector)	cannot write to any other elements in Vector
for (v <- Vertices)	can only write to neighboring vertices, e.g. v.neighbors
for (e <- Matrix)	cannot write to any other elements in Matrix
for (r <- Matrix.rows)	cannot write to any other rows in Matrix
for (p <- Image)	can only write to pixels in a local window, e.g. p.window
for (r <- Stream.rows)	cannot access any other rows in Stream

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:

```
untilconverged[A](x: Rep[A], thresh: Rep[Double], max_iter: Rep[Int] = unit(1000),
                  clone_prev_val: Rep[Boolean] = unit(false))
                 (block: Rep[A] => Rep[A])
                 (implicit diff: (Rep[A],Rep[A]) => Rep[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:

```
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:

```
batch(x: Rep[TrainingSet[Double]], alpha: Rep[Double], thresh: Rep[Double], maxIter: Rep[Int])
     (hyp: Rep[Vector[Double]] => Rep[Double]): 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:

```
block() updates each jth parameter from the sum of all ith training samples
while not converged
  for j from 0 until n
    j_update = sum((y(i) - h(x(i))*x(j,i)
    updatej(j_update)
```

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

```
block() updates every jth parameter for every ith training sample
while not converged
  for i from 0 until m
    for j from 0 until n
      updatej(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

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†.

Below is an example of computing a simple scalar sum:

```
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:

```
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:

```
// 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) }
```

†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 row IndexVector and a col IndexVector†:

```
// 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:

```
// 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:

```
// 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).

†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 matric arguments to the dist function, with an optional metric parameter (the default metric is ABS):

```
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. $|d1 - d2|$:

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

6.4 det

det computes the determinant of the input matrix m:

```
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.

```
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 `ABS`) to compute the nearest neighbor.

```
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.[5]

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:

```
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:

```
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:

```
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 specifiying a different delimeter than the default tab.

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

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

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

// uses a semicolon delimeter, 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

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:

```
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-delimeted 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 `v1+v2*s/100`.

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

```
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) * X(j)).sum` and op fusing will ensure that no intermediate vector will be allocated to hold the result of `X(i) * 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

```
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[6] [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 `untilconverged` construct that allows some number of iterations to be run in parallel. Recent work [10] has shown the potential for this optimization.

[6] These 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 *Rep* type. For example, one could specify two illustrative type-bounds on methods as follows:

```
def foo[A <: Double](x: Rep[Vector[A]]) { ... }
def bar[B <: Rep[Vector[Double]]](x: B) { ... }
```

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 [5].

8.2 Pass by value

OptiML also follows the Scala/Java pass by value semantics, where objects are *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 `Vector[Foo]`, returning `v(0)` will return a reference to the `Foo` object at element 0. Similarly, accessing row `i` of a Matrix using `m(i)` returns a reference to a new `VectorView` representing the elements at that position. Accessing a primitive object, e.g. `v(0)` in a `Vector[Double]` will return the value of the double of at `v(0)`.

8.3 Mutable/Immutable operators

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

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

8.4 Large scale program organization

All OptiML code must be executed within the dynamic scope of the `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 `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 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`:

```
// 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`:

```
// we can even use math operators by defining how arithmetic works with MyStruct
implicit def myStructArith: Arith[MyStruct] = new Arith[MyStruct] {
  def +=(a: Rep[MyStruct], b: Rep[MyStruct])(implicit ctx: SourceContext) =
    newMyStruct(a.data+b.data,a.name)
  def +(a: Rep[MyStruct], b: Rep[MyStruct])(implicit ctx: SourceContext) =
    newMyStruct(a.data+b.data,a.name+" pl "+b.name)
  def -(a: Rep[MyStruct], b: Rep[MyStruct])(implicit ctx: SourceContext) =
    newMyStruct(a.data-b.data,a.name+" mi "+b.name)
  def *(a: Rep[MyStruct], b: Rep[MyStruct])(implicit ctx: SourceContext) =
    newMyStruct(a.data*b.data,a.name+" ti "+b.name)
  def /(a: Rep[MyStruct], b: Rep[MyStruct])(implicit ctx: SourceContext) =
    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:

```
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

[1] Kevin J. Brown, Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky & Kunle Olukotun (2011): *A Heterogeneous Parallel Framework for Domain-Specific Languages*. PACT.

[2] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Y. Ng & K. Olukotun (2007): *Map-Reduce for Machine Learning on Multicore*. In: *Advances in Neural Information Processing Systems 19*.

[3] M. Kearns (1998): *Efficient noise-tolerant learning from statistical queries*. Joural of the ACM 45, pp. 983–1006.

[4] J. Meng, S. Chakradhar & A. Raghunathan (2009): *Best-Effort Parallel Execution Framework for Recognition and Mining Applications*. In: *Proc. of IPDPS*.

[5] M. Odersky (2011): *Scala*. `http://www.scala-lang.org`.

[6] J. C. Platt (1998): *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines*.

[7] Tiark Rompf & Martin Odersky (2010): *Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs*. In: *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE, ACM, New York, NY, USA, pp. 127–136, doi:http://doi.acm.org/10.1145/1868294.1868314. Available at `http://doi.acm.org/10.1145/1868294.1868314`.

[8] Tiark Rompf, Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Martin Odersky & Kunle Olukotun (2011): *Building-Blocks for Performance Oriented DSLs*. DSL, doi:http://dx.doi.org/10.4204/EPTCS.

[9] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, Michael Wu, A. R. Atreya, M. Odersky & K. Olukotun (2011): *OptiML: an Implicitly Parallel Domain-Specific Language for Machine Learning*. In: *Proceedings of the 28th International Conference on Machine Learning*, ICML.

[10] M. A. Zinkevich, M. Weimer, A. Smola & L. Li (2010): *Parallelized Stochastic Gradient Descent*. In: *Advances in Neural Information Processing Systems*.