OptiML: An Implicitly Parallel Domain-Specific Language for ML

Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Michaël Wu, Anand Atreya, Kunle Olukotun
Stanford University
Pervasive Parallelism Laboratory (PPL)

Tiark Rompf, Martin Odersky
Ecole Polytechnique Federale de Lausanne (EPFL)
Programming Methods Laboratory
Machine Learning

- Learning patterns from data
  - Regression
  - Classification (e.g. SVMs)
  - Clustering (e.g. K-Means)
  - Density estimation (e.g. Expectation Maximization)
  - Inference (e.g. Loopy Belief Propagation)
  - Adaptive (e.g. Reinforcement Learning)

- A good domain for studying parallelism
  - Many applications and datasets are time-bound in practice
  - A combination of regular and irregular parallelism at varying granularities
  - At the core of many emerging applications (speech recognition, robotic control, data mining etc.)
Machine Learning Applications

Finding movies you'll ❤️ just got easier...

Rate a few movies you've seen and we can help you find movies you'll enjoy.

The more you rate, the smarter Netflix becomes... making it easier to find that hidden gem you may have missed or forgotten about.

Continue

It just takes 2 minutes...

Report Spam
Example algorithms

Computing parameters:

- Naïve Bayes \[ \sum_{i=1}^{n} \log(P(x_i | y=1)) + \log(P(y=1)) \]
- GDA \[ E = \frac{1}{m} \sum_i (x^{(i)} - \mu_{y(i)}) (x^{(i)} - \mu_{y(i)})^T \]

Iterative convergence:

- linear regression (gradient descent)
- Newton’s method (numerical approximation)

Data manipulation:

- collaborative filtering (group, map)
- image processing (slicing, filtering, searching)
DESIGNING DSLS: REQUIRED EXPERTISE
Major Challenges

- Expressing the important problems
- Elegant, natural and simple design
- Implementing efficiently and portably
Domain Expertise

Expressing the important problems

Gradient Descent
Convex Optimization
Linear Algebra

Images, Video, Audio
Probabilistic Message-passing graphs
Streaming training sets
Language Expertise

Abstract Syntax Tree
Control Flow Graph

Program Transformation
Alias Analysis

Code Generation
Loop-invariant Code Motion

Elegant, natural and simple design
Performance Expertise

Thread

False Sharing

Locality

Mutex

SSE  Synchronization

Coherency Protocol

Synchronization

TLB Shootdown  Bandwidth

Implementing efficiently and portably
DSL Implementations

- **Stand-alone**
  - Domain expertise, language expertise and performance expertise

- **Embedded in a host language**
  - Domain expertise and performance expertise

- **Embedded with a common framework**
  - DSL authors focus mainly on domain expertise
  - Framework authors provide language and performance expertise

Delite
OptiML: Approach

- Identify high-level abstractions common in ML

- Provide those abstractions as first-class data types or functional operators

- Use knowledge of those operators to optimize and generate efficient, imperative code
OptiML: Overview

- Provides a familiar (MATLAB-like) language and API for writing ML applications
  - Ex. `val c = a * b` (a, b are Matrix[Double])

- Implicitly parallel data structures
  - Base types
    - Vector[T], Matrix[T], Graph[V,E], Stream[T]
  - Subtypes
    - TrainingSet, IndexVector, Image, ...

- Implicitly parallel control structures
  - `sum{...}`, `(0::end) {...}`, `gradient { ... }`, `untilconverged { ... }`
  - Allow anonymous functions with restricted semantics to be passed as arguments of the control structures
Newton’s Method in OptiML

// f, df, x0, tol, nmax inputs
var x = x0 - (f(x0)/df(x0)) // approximation to root
var ex = abs(x-x0)         // error estimate
until converged(ex, tol) { ex =>
    val x2 = x - (f(x)/df(x))
    val err = abs(x-x2)
    x = x2
    err
}
OptiML: Implementation

OptiML program

build, analyze, optimize intermediate representation

eDSL Compiler implemented with Delite framework

Delite Execution Graph

Scala ops

CUDA ops

Other targets

Scheduling

Address space management

Communication/Synchronization

Delite runtime
OptiML: Advantages

- **Productive**
  - Operate at a higher level of abstraction
  - Focus on algorithmic description, get parallel performance

- **Portable**
  - Single source => Multiple heterogeneous targets
  - Not possible with today’s MATLAB support

- **High Performance**
  - Builds and optimizes an intermediate representation (IR) of programs
  - Generates efficient code specialized to each target
Manipulating Vectors and Matrices

val a = Vector(1,2,3,4,5)
val b = Matrix(a,Vector(4,5,6,7,8))

val c = (0::100) { i => i*2 }
val d = (0::10,0::10) { (i,j) => i*j }
val e = (0::100,*) { i =>
    Vector.rand(10)
}

val f = b*a.t+(c.slice(0,2)*log(2)).t
(f map { e => e + 2 }).min
k-Means Clustering

until converged(mu, tol){
    mu =>
    // calculate distances to current centroids
    // move each cluster centroid to the
    // mean of the points assigned to it
}
k-Means Clustering

```scala
until_converged(mu, tol){
  mu =>
  // calculate distances to current centroids
  val c = (0::m){i =>
    val allDistances = mu.map_rows { centroid =>
      dist(x(i), centroid)
    }
    allDistances.min_index
  }

  // move each cluster centroid to the
  // mean of the points assigned to it
}
```
**k-Means Clustering**

```scala
untilConverged(mu, tol) { mu =>
  // calculate distances to current centroids
  val c = (0::m)(i =>
    val allDistances = mu mapRows { centroid =>
      dist(x(i), centroid)
    }
    allDistances.minIndex
  }

  // move each cluster centroid to the mean of the points assigned to it
  val newMu = (0::k, *) { i =>
    val (weightedPoints, points) = sum(0, m) { j =>
      if (c(i) == j) (x(i), 1)
    }
    if (points == 0) Vector.zeros(n) else weightedPoints / points
  }
  newMu
}
```
# OptiML vs. MATLAB

<table>
<thead>
<tr>
<th>OptiML</th>
<th>MATLAB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statically typed</td>
<td>Dynamically typed</td>
</tr>
<tr>
<td>No explicit parallelization</td>
<td>Applications must explicitly choose between vectorization or parallelization</td>
</tr>
<tr>
<td>Automatic GPU data management via run-time support</td>
<td>Explicit GPU data management</td>
</tr>
<tr>
<td>Inherits Scala features and tool-chain</td>
<td>Widely used, numerous libraries and toolboxes</td>
</tr>
<tr>
<td>Machine learning specific abstractions</td>
<td></td>
</tr>
</tbody>
</table>
MATLAB parallelism

- `parfor` is nice, but not always best
  - MATLAB uses heavy-weight MPI processes under the hood
  - Precludes vectorization, a common practice for best performance
  - GPU code requires different constructs

- The application developer must choose an implementation, and these details are all over the code

```matlab
ind = sort(randsample(1:size(data,2),length(min_dist)));
data_tmp = data(:,ind);
all_dist = zeros(length(ind),size(data,2));
parfor i=1:size(data,2)
    all_dist(:,i) =
        sum(abs(repmat(data(:,i),1,size(data_tmp,2)) - data_tmp),1)';
end
all_dist(all_dist==0)=max(max(all_dist));
```
OptiML is Declarative and Restricted

- Allows only a small subset of Scala

- User-defined data structures must be structs (no methods)

- Anonymous functions cannot have side effects

```scala
val c = (0::m) {e => /* pure! */ (no disjoint writes!)

- Object instances cannot be mutated unless .mutable is called first

```scala
val v = Vector(1,2,3,4)
v(0) = 5 // compile error!

```scala
val v2 = v.mutable
v2(0) = 5
```

OptiML does not have to be conservative

Guarantees major properties (e.g. parallelizable) by construction
OptiML Optimizations

- Common subexpression elimination (CSE), Dead code elimination (DCE), Code motion
- Pattern rewritings
  - Linear algebra simplifications
  - Shortcuts to help fusing
- Op fusing
  - can be especially useful in ML due to fine-grained operations and low arithmetic intensity

Coarse-grained: optimizations happen on vectors and matrices
A straightforward translation of the Gaussian Discriminant Analysis (GDA) algorithm from the mathematical description produces the following code:

```
val sigma = sum(0,m) { i =>
  if (x.labels(i) == false) {
    ((x(i) - mu0).t) ** (x(i) - mu0)
  } else
    ((x(i) - mu1).t) ** (x(i) - mu1)
}
```

A much more efficient implementation recognizes that

\[
\sum_{i=0}^{n} \vec{x}_i \cdot \vec{y}_i \rightarrow \sum_{i=0}^{n} X(:,i) \cdot Y(i,:) = X \cdot Y
\]

Transformed code was **20.4x** faster with 1 thread and **48.3x** faster with 8 threads.
Putting it all together: SPADE

Downsample:
L1 distances between all $10^6$ events in 13D space... reduce to 50,000 events

```scala
val distances = Stream[Double](data.numRows, data.numRows){
  (i,j) => dist(data(i),data(j))
}

for (row <- distances.rows) {
  if(densities(row.index) == 0) {
    val neighbors = row find { _ < apprXWidth }
    densities(neighbors) = row count { _ < kernelWidth }
  }
}
```
val distances = Stream[Double](data.numRows, data.numRows) {
  (i,j) => dist(data(i),data(j))
}

for (row <- distances.rows) {
  row.init // expensive! part of the stream foreach operation
  if (densities(row.index) == 0) {
    val neighbors = row find { _ < apprxWidth }
    densities(neighbors) = row count { _ < kernelWidth }
  }
}

row is 235,000 elements in one typical dataset – fusing is a big win!
// FOR EACH ELEMENT IN ROW
while (x155 < x61) {
    val x168 = x155 * x64
    var x180 = 0

    // INITIALIZE STREAM VALUE (dist(i,j))
    while (x180 < x64) {
        val x248 = x164 + x180
        // . . .
    }
}

// VECTOR FIND
if (x245) x201.insert(x201.length, x155)

// VECTOR COUNT
if (x246) {
    val x207 = x208 + 1; x208 = x207
}
val x207 = x208 + 1; x208 = x207
x155 += 1

From a ~5 line algorithm description in OptiML

...to an efficient, fused, imperative version that closely resembles a hand-optimized C++ baseline!
Performance Results

Machine
- Two quad-core Nehalem 2.67 GHz processors
- NVidia Tesla C2050 GPU

Application Versions
- OptiML + Delite
- MATLAB
  - version 1: multi-core (parallelization using “parfor” construct and BLAS)
  - version 2: MATLAB GPU support
  - version 3: Accelereyes Jacket GPU support
- C++
  - Optimized reference baselines for larger applications
Experiments on ML kernels

- **GDA**
- **Naive Bayes**
- **K-means**
- **SVM**
- **Linear Regression**
- **RBM**

**OptiML**  **Parallelized MATLAB**  **MATLAB + Jacket**
Experiments on larger apps

<table>
<thead>
<tr>
<th>Normalized Execution Time</th>
<th>OptiML</th>
<th>C++</th>
<th>OptiML</th>
<th>C++</th>
<th>OptiML</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 CPU</td>
<td>1.0</td>
<td>0.7</td>
<td>1.0</td>
<td>0.9</td>
<td>1.0</td>
<td>0.9</td>
</tr>
<tr>
<td>2 CPU</td>
<td>1.7</td>
<td></td>
<td>1.7</td>
<td></td>
<td>1.7</td>
<td></td>
</tr>
<tr>
<td>4 CPU</td>
<td>3.1</td>
<td></td>
<td>1.5</td>
<td></td>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td>8 CPU</td>
<td>4.9</td>
<td></td>
<td>3.3</td>
<td></td>
<td>5.4</td>
<td></td>
</tr>
</tbody>
</table>

Different applications and CPU counts are shown in the diagram.
Impact of Op Fusion

Normalized Execution Time vs. Processors

- C++
- OptiML Fusing
- OptiML No Fusing
Summary

- OptiML is a proof-of-concept DSL for ML embedded in Scala using the Delite framework.
- OptiML translates simple, declarative machine learning operations to optimized code for multiple platforms.
- Outperforms MATLAB and C++ on a set of well-known machine learning applications.
Thank you!

- Find us on Github:
  - https://github.com/stanford-ppl/Delite/optiml

- Mailing list
  - http://groups.google.com/group/optiml

- Comments and criticism welcome

- Questions?
backup
OptiML: Approach

- Encourage a functional, parallelizable style through restricted semantics
  - Fine-grained, composable map-reduce operators

- Map ML operations to parallel operations (domain decomposition)

- Automatically synchronize parallel iteration over domain-specific data structures
  - Exploit structured communication patterns (nodes in a graph may only access neighbors, etc.)

- Defer as many implementation-specific details to compiler and runtime as possible

OptiML does not have to be conservative

Guarantees major properties (e.g. parallelizable) by construction
Example OptiML / MATLAB code (Gaussian Discriminant Analysis)

```
val sigma = sum(i, x.numSamples) {
  if (x.labels(i) == false) {
    (x(i) - mu0).trans.outer(x(i) - mu0)
  } else {
    (x(i) - mu1).trans.outer(x(i) - mu1)
  }
}
```

```
parfor i=1:length(y)
  if (y(i) == 0)
    sigma = sigma + (x(i,:) - mu0)'*(x(i,:) - mu0);
  else
    sigma = sigma + (x(i,:) - mu1)'*(x(i,:) - mu1);
  end
end
```

ML-specific data types
- Implicitly parallel control structures
- Restricted index semantics

% x : Matrix, y: Vector
% mu0, mu1: Vector
n = size(x,2);
sigma = zeros(n,n);

OptiML code

(parallel) MATLAB code
Experiments on ML kernels (C++)

- **OptiML**

### GDA

<table>
<thead>
<tr>
<th>Normalized Execution Time</th>
<th>1 CPU</th>
<th>2 CPU</th>
<th>4 CPU</th>
<th>8 CPU</th>
<th>CPU + GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 CPU</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2 CPU</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>4 CPU</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
</tr>
<tr>
<td>8 CPU</td>
<td>4.1</td>
<td>4.1</td>
<td>4.1</td>
<td>4.1</td>
<td>4.1</td>
</tr>
</tbody>
</table>

### Naive Bayes

<table>
<thead>
<tr>
<th>Normalized Execution Time</th>
<th>1 CPU</th>
<th>2 CPU</th>
<th>4 CPU</th>
<th>8 CPU</th>
<th>CPU + GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 CPU</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2 CPU</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>4 CPU</td>
<td>1.9</td>
<td>1.9</td>
<td>1.9</td>
<td>1.9</td>
<td>1.9</td>
</tr>
<tr>
<td>8 CPU</td>
<td>3.6</td>
<td>3.6</td>
<td>3.6</td>
<td>3.6</td>
<td>3.6</td>
</tr>
</tbody>
</table>

### K-means

<table>
<thead>
<tr>
<th>Normalized Execution Time</th>
<th>1 CPU</th>
<th>2 CPU</th>
<th>4 CPU</th>
<th>8 CPU</th>
<th>CPU + GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 CPU</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2 CPU</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>4 CPU</td>
<td>2.1</td>
<td>2.1</td>
<td>2.1</td>
<td>2.1</td>
<td>2.1</td>
</tr>
<tr>
<td>8 CPU</td>
<td>4.1</td>
<td>4.1</td>
<td>4.1</td>
<td>4.1</td>
<td>4.1</td>
</tr>
</tbody>
</table>

### SVM

<table>
<thead>
<tr>
<th>Normalized Execution Time</th>
<th>1 CPU</th>
<th>2 CPU</th>
<th>4 CPU</th>
<th>8 CPU</th>
<th>CPU + GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 CPU</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2 CPU</td>
<td>1.9</td>
<td>1.9</td>
<td>1.9</td>
<td>1.9</td>
<td>1.9</td>
</tr>
<tr>
<td>4 CPU</td>
<td>3.1</td>
<td>3.1</td>
<td>3.1</td>
<td>3.1</td>
<td>3.1</td>
</tr>
<tr>
<td>8 CPU</td>
<td>4.2</td>
<td>4.2</td>
<td>4.2</td>
<td>4.2</td>
<td>4.2</td>
</tr>
</tbody>
</table>

### Linear Regression

<table>
<thead>
<tr>
<th>Normalized Execution Time</th>
<th>1 CPU</th>
<th>2 CPU</th>
<th>4 CPU</th>
<th>8 CPU</th>
<th>CPU + GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 CPU</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2 CPU</td>
<td>1.4</td>
<td>1.4</td>
<td>1.4</td>
<td>1.4</td>
<td>1.4</td>
</tr>
<tr>
<td>4 CPU</td>
<td>1.4</td>
<td>1.4</td>
<td>1.4</td>
<td>1.4</td>
<td>1.4</td>
</tr>
<tr>
<td>8 CPU</td>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>

### RBM

<table>
<thead>
<tr>
<th>Normalized Execution Time</th>
<th>1 CPU</th>
<th>2 CPU</th>
<th>4 CPU</th>
<th>8 CPU</th>
<th>CPU + GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 CPU</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2 CPU</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>4 CPU</td>
<td>1.4</td>
<td>1.4</td>
<td>1.4</td>
<td>1.4</td>
<td>1.4</td>
</tr>
<tr>
<td>8 CPU</td>
<td>2.3</td>
<td>2.3</td>
<td>2.3</td>
<td>2.3</td>
<td>2.3</td>
</tr>
</tbody>
</table>

- **CPU**
- **GPU**

---
Dynamic Optimizations

- Relaxed dependencies
  - Iterative algorithms with inter-loop dependencies prohibit task parallelism
  - Dependencies can be relaxed at the cost of a marginal loss in accuracy
  - Relaxation percentage is run-time configurable

- Best effort computations
  - Some computations can be dropped and still generate acceptable results
  - Provide data structures with “best effort” semantics, along with policies that can be chosen by DSL users
Dynamic optimizations

K-means Best Effort

Normalized Execution Time

SVM Relaxed Dependencies

- K-means
- Best-effort (1.2% error)
- Best-effort (4.2% error)
- Best-effort (7.4% error)

- SVM
- Relaxed SVM (+ 1% error)