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
Background

- We are researchers in programming languages, parallel programming, and computer architecture

- Working with machine learning and bioinformatics groups at Stanford and elsewhere

- Would love to work with you and get your feedback, suggestions, and criticism
Heterogeneous Parallel Programming

- Pthreads
- OpenMP
- CUDA
- OpenCL
- Verilog
- VHDL
- MPI
- Sun T2
- Nvidia Fermi
- Altera FPGA
- Cray Jaguar
Programmability Chasm

Too many different programming models

Applications

- Scientific Engineering
- Virtual Worlds
- Personal Robotics
- Data informatics

Tools and Technologies

- Pthreads
- OpenMP
- CUDA
- OpenCL
- Verilog
- VHDL
- MPI
- Sun T2
- Nvidia Fermi
- Altera FPGA
- Cray Jaguar
IS IT POSSIBLE TO WRITE ONE PROGRAM AND RUN IT ON ALL THESE TARGETS?
HYPOTHESIS: YES, BUT NEED DOMAIN-SPECIFIC LANGUAGES
The Ideal Parallel Programming Language

Performance

Productivity  Generality
Successful Languages

Performance

Productivity

Generality

C/C++

Java

Python

Ruby
Successful Languages

Performance

- DSLs

Productivity

- C/C++

Generality

- Python

- Ruby

- Java
OptiML: A DSL For ML

- **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
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
  - General data types: Vector[T], Matrix[T], Graph[V,E]
    - Independent from the underlying implementation
  - Specialized data types: Stream, TrainingSet, TestSet, IndexVector, Image, Video ..
    - Encode semantic information & structured, synchronized communication

- Implicitly parallel control structures
  - `sum{...}, (0::end) {...}, gradient { ... }, untilconverged { ... }`
  - Allow anonymous functions with restricted semantics to be passed as arguments of the control structures
**OptiML: K-means example**

```scala
untilConverged(mu, tol) { mu =>
  // calculate distances to current centroids
  val c = (0::m){i =>
    val allDistances = mu mapRows { centroid =>
      // distance from sample x(i) to centroid
      ((x(i) - centroid) * (x(i) - centroid)).sum
    } -> c
    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)
      else
        0
    }
    if (points == 0) Vector.zeros(n)
    else weightedpoints / points
  }
  newMu
}
```

*Multiple granularities of parallelism*

*Normal matrix/vector arithmetic syntax*

*Control structure can only access indices i and j (disjoint)*
OptiML vs. MATLAB

**OptiML**
- Statically typed
- No explicit parallelization
- Automatic GPU data management via run-time support
- Inherits Scala features and tool-chain
- Machine learning specific abstractions

**MATLAB**
- Dynamically typed
- Applications must explicitly choose between vectorization or parallelization
- Explicit GPU data management
- Widely used, numerous libraries and toolboxes
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(randn(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 Implementation

- eDSL Compiler implemented with Delite framework
  - build, analyze, optimize intermediate representation

- Delite Execution Graph
- Scala ops
- CUDA ops
- Other targets

- Scheduling
- Address space management
- Communication/Synchronization
- Delite runtime
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:

```scala
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

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

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 }
    }
}
```
**SPADE transformations**

```scala
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!
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

OptiML  Parallelized MATLAB  MATLAB + Jacket

GDA

Normalized Execution Time

Naive Bayes

K-means

SVM

Linear Regression

RBM
Experiments on larger apps

<table>
<thead>
<tr>
<th></th>
<th>OptiML</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>TM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 CPU</td>
<td>1.0</td>
<td>0.7</td>
</tr>
<tr>
<td>2 CPU</td>
<td>1.7</td>
<td></td>
</tr>
<tr>
<td>4 CPU</td>
<td>3.1</td>
<td></td>
</tr>
<tr>
<td>8 CPU</td>
<td>4.9</td>
<td></td>
</tr>
<tr>
<td>LBP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 CPU</td>
<td>1.0</td>
<td>1.2</td>
</tr>
<tr>
<td>2 CPU</td>
<td>1.7</td>
<td>1.5</td>
</tr>
<tr>
<td>4 CPU</td>
<td>2.5</td>
<td>3.5</td>
</tr>
<tr>
<td>8 CPU</td>
<td>3.3</td>
<td>5.4</td>
</tr>
<tr>
<td>SPADE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 CPU</td>
<td>1.0</td>
<td>0.9</td>
</tr>
<tr>
<td>2 CPU</td>
<td>1.9</td>
<td>1.8</td>
</tr>
<tr>
<td>4 CPU</td>
<td>3.4</td>
<td>3.3</td>
</tr>
<tr>
<td>8 CPU</td>
<td>5.8</td>
<td>5.6</td>
</tr>
</tbody>
</table>
Impact of Op Fusion

Normalized Execution Time

- C++
- OptiML Fusing
- OptiML No Fusing

Processors

- 1
  - C++: 0.9
  - OptiML Fusing: 1.0
  - OptiML No Fusing: 0.3

- 2
  - C++: 1.8
  - OptiML Fusing: 1.9
  - OptiML No Fusing: 0.6

- 4
  - C++: 3.3
  - OptiML Fusing: 3.4
  - OptiML No Fusing: 0.9

- 8
  - C++: 5.6
  - OptiML Fusing: 5.8
  - OptiML No Fusing: 1.0
Summary

- DSLs are a promising parallel programming platform
  - Capable of achieving portability, productivity, and high performance

- OptiML is a proof-of-concept DSL for ML embedded in Scala, using the Lightweight Modular Staging (LMS) framework and Delite

- 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!

- For the brave, find us on Github: 
  - https://github.com/stanford-ppl/Delite
  - (very alpha)

- Comments and criticism very 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.)
- OptiML does not have to be conservative
  - Guarantees major properties (e.g. parallelizable) by construction
- Defer as many implementation-specific details to compiler and runtime as possible
Example OptiML / MATLAB code (Gaussian Discriminant Analysis)

```
// x : TrainingSet[Double]
// mu0, mu1 : Vector[Double]
val sigma = sum(0,x.numSamples) {
  if (x.labels(_ == false) {
    (x(_)-mu0).trans.outer(x(_)-mu0)
  }
  else {
    (x(_)-mu1).trans.outer(x(_)-mu1)
  }
}

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

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

OptiML code

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

![Graphs showing comparison of execution times for different algorithms and configurations: GDA, Naive Bayes, K-means, SVM, Linear Regression, and RBM. Each graph compares OptiML, Parallelized MATLAB, and C++ execution times across 1, 2, 4, 8 CPUs, and CPUs + GPU combinations.]
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

SVM Relaxed Dependencies

Normalized Execution Time

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

- SVM
- Relaxed SVM (+ 1% error)