CS442: High Productivity and Performance with Domain-specific Languages in Scala
CS442 Overview

- Experimental Research Course
  - Lots of interest in DSLs
  - Opportunity to do something new
  - New ideas: DSLs for parallelism and productivity
  - New software: DSL infrastructure
  - May lead to publications

- Goals
  - Knowledge and tools for DSL development
  - Bring domain experts and CS experts together
  - Develop new DSLs and push development of DSL infrastructure
  - Learn Scala: important new programming lang.
Course Information

- **Instructor:** Kunle Olukotun
  - E-mail: kunle@stanford.edu
  - Office: Gates 302
  - Office Hours: 11:00am-noon M/W or by appointment

- **TAs**
  - Hassan Chafi, Arvind Sujeeth, Kevin Brown
  - E-Mail: cs442-spr1011-staff at lists dot stanford dot edu

- **Course Support**
  - Darlene Hadding
  - E-Mail: darleneh at stanford dot edu
  - Location: Gates 408
  - Telephone: 650-723-1430
  - Office Hours: M-F 10:00 AM-3:00 PM
Course Information

- **Prerequisites**
  - CS background: CS108, and a systems course (CS143, CS140), preferably CS 149.
  - Non CS background: Expertise in a particular domain

- **Participation (10% of grade)**
  - You are expected to participate in the class discussion

- **Website**
  - [http://www.stanford.edu/class/cs442](http://www.stanford.edu/class/cs442)

- **Programming Assignments**
  - Scala
  - DSL embedding with

- **Final Project (90% of grade)**
  - The Final Project is an open-ended research project
  - New DSL or adapt existing DSL
2020 Vision for Parallelism

- Make parallelism accessible to all programmers
- Parallelism is not for the average programmer
  - Too difficult to find parallelism, to debug, maintain and get good performance for the masses
  - Need a solution for “Joe/Jane the programmer”
- Can’t expose average programmers to parallelism
  - But auto parallelization doesn’t work
What Is Computing?

- **Predicting the future**
  - Modeling and simulation (weather, materials, products)
  - Decide what to build and experiment or instead of build and experiment ⇒ third pillar of science

- **Coping with the present (real time)**
  - Embedded systems control (cars, planes, communication)
  - Virtual worlds (second life, facebook)
  - Electronic trading (airline reservation, stock market)
  - Robotics (manufacturing, cars, household)

- **Understanding the past**
  - Large data set analysis (commerce, web, census, simulation)
  - Discover trends and develop insight
# The Changing Nature of Science

<table>
<thead>
<tr>
<th>Experimental</th>
<th>Theoretical</th>
<th>Computational</th>
<th>The Fourth Paradigm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thousand years ago</td>
<td>Last few hundred years</td>
<td>Last few decades</td>
<td>Today and the Future</td>
</tr>
<tr>
<td>Description of natural phenomena</td>
<td>Newton’s laws, Maxwell’s equations...</td>
<td>Simulation of complex phenomena</td>
<td>Unify theory, experiment and simulation with large multidisciplinary data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Using data exploration and data mining (from instruments, sensors, humans...)</td>
</tr>
</tbody>
</table>
Explosion of Data Sources

The Challenge
Enable Discovery

Deliver the capability to mine, search and analyze this data in near real time

Petabytes Doubling & Doubling

The Response
Discovery itself is evolving

Experiments
Simulations
Archives
Literature
Consumer

(C) 2011 Kunle Olukotun
Genetics Gets Personal

CMOS Sensing and Computing

$3 billion per Genome

$45,000 per Genome

$500-$10,000 per Genome

$100 per Genome?

3e09 bytes per person
x 6e09 people
= 1.8e19 bytes
= 10K petabytes
Computing System Power

\[ \text{Power} = \text{Energy} \times \frac{\text{Ops}}{\text{second}} \]

**FIXED**
Heterogeneous Hardware

- Heterogeneous HW for energy efficiency
  - Multi-core, ILP, threads, data-parallel engines, custom engines

- H.264 encode study

Source: Understanding Sources of Inefficiency in General-Purpose Chips (ISCA’10)
DE Shaw Research: Anton

Molecular dynamics computer

100 times more power efficient

D. E. Shaw et al. SC 2009, Best Paper and Gordon Bell Prize
Heterogeneous Parallel Architectures Today

Sun T2
Nvidia Fermi
Altera FPGA
Cray Jaguar
Heterogeneous Parallel Programming

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

Too many different programming models
Hypothesis

It is possible to write one program and run it on all these machines
Programmability Chasm

Applications

Scientific Engineering
Virtual Worlds
Personal Robotics
Data informatics

Ideal Parallel Programming Language

Pthreads
OpenMP
CUDA
OpenCL
Verilog
VHDL
MPI
PGAS

Sun T2
Nvidia Fermi
Altera FPGA
Cray Jaguar

(C) 2011 Kunle Olukotun
The Ideal Parallel Programming Language

Performance

Productivity

Generality
Successful Languages

Performance

Productivity

Generality

C/C++

Python

Ruby

(C) 2011 Kunle Olukotun
True Hypothesis $\Rightarrow$ Domain Specific Languages

Performance
(Heterogeneous Parallelism)

Domain Specific Languages

Productivity

Generality

\[ \text{C/C++} \]

\[ \text{Java} \]

\[ \text{python} \]

\[ \text{Ruby} \]
Domain Specific Languages

- Domain Specific Languages (DSLs)
  - Programming language with restricted expressiveness for a particular domain
  - High-level, usually declarative, and deterministic

![OpenGL](image1.png)
![MATLAB](image2.png)
![MySQL](image3.png)
![RAILS](image4.png)
![TEX](image5.png)

(C) 2011 Kunle Olukotun
Benefits of Using DSLs for Parallelism

Productivity
- Shield average programmers from the difficulty of parallel programming
- Focus on developing algorithms and applications and not on low level implementation details

Performance
- Match high level domain abstraction to generic parallel execution patterns
- Restrict expressiveness to more easily and fully extract available parallelism
- Use domain knowledge for static/dynamic optimizations

Portability and forward scalability
- DSL & Runtime can be evolved to take advantage of latest hardware features
- Applications remain unchanged
- Allows innovative HW without worrying about application portability

(C) 2011 Kunle Olukotun
Bridging the Programmability Chasm

Applications

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

Domain Specific Languages

- Rendering
- Physics (Liszt)
- Data Analysis (SQL)
- Probabilistic (RandomT)
- Machine Learning (OptiML)

Domain Embedding Language (Scala)

- Polymorphic Embedding
- Staging
- Static Domain Specific Opt.

Parallel Runtime (Delite)

- Task & Data Parallelism
- Locality Aware Scheduling

Heterogeneous Hardware

(C) 2011 Kunle Olukotun
Liszt: DSL for Mesh PDEs

- Z. DeVito, N. Joubert, P. Hanrahan
- Solvers for mesh-based PDEs
  - Complex physical systems
  - Huge domains
  - millions of cells
  - Example: Unstructured Reynolds-averaged Navier Stokes (RANS) solver
- Goal: simplify code of mesh-based PDE solvers
  - Write once, run on any type of parallel machine
  - From multi-cores and GPUs to clusters

(C) 2011 Kunle Olukotun
Liszt Language Features

- Minimal Programming language
  - Arithmetic, short vectors, functions, control flow

- Built-in mesh interface for arbitrary polyhedra
  - Vertex, Edge, Face, Cell
  - Optimized memory representation of mesh

- Collections of mesh elements
  - Element Sets: faces(c:Cell), edgesCCW(f:Face)

- Mapping mesh elements to fields
  - Fields: val vert_position = position(v)

- Parallelizable iteration
  - forall statements: for( f <- faces(cell) ) { … }
Liszt Code Example

for(edge <- edges(mesh)) {
    val flux = flux_calc(edge)
    val v0 = head(edge)
    val v1 = tail(edge)
    Flux(v0) += flux
    Flux(v1) -= flux
}

Simple Set Comprehension
Functions, Function Calls
Mesh Topology Operators
Field Data Storage

Code contains possible write conflicts!
We use architecture specific strategies
guided by domain knowledge
- MPI: Ghost cell-based message passing
- GPU: Coloring-based use of shared memory
MPI Performance

- Using 8 cores per node, scaling up to 96 cores (12 nodes, 8 cores per node, all communication using MPI)

**MPI Speedup 750k Mesh**

<table>
<thead>
<tr>
<th>Number of MPI Nodes</th>
<th>Speedup over Scalar</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>120</td>
<td>120</td>
</tr>
</tbody>
</table>

**Linear Scaling**

**Liszt Scaling**

**Joe Scaling**

**MPI Wall-Clock Runtime**

<table>
<thead>
<tr>
<th>Number of MPI Nodes</th>
<th>Runtime Log Scale (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>40</td>
<td>10</td>
</tr>
<tr>
<td>60</td>
<td>1</td>
</tr>
<tr>
<td>80</td>
<td>0.1</td>
</tr>
<tr>
<td>100</td>
<td>0.01</td>
</tr>
<tr>
<td>120</td>
<td>0.001</td>
</tr>
</tbody>
</table>

**Liszt Runtime**

**Joe Runtime**
GPU Performance

- Scaling mesh size from 50K (unit-sized) cells to 750K (16x) on a Tesla C2050. Comparison is against single threaded runtime on host CPU (Core 2 Quad 2.66Ghz)

Single-Precision: 31.5x, Double-precision: 28x
OptiML: A DSL for ML

- A. Sujeeth and H. Chafi
- Machine Learning domain
  - Learning patterns from data
  - Applying the learned models to tasks
    - Regression, classification, clustering, estimation
  - Computationally expensive
  - Regular and irregular parallelism

Motivation for OptiML

- Raise the level of abstraction
- Use domain knowledge to identify coarse-grained parallelism
- Single source ⇒ multiple heterogeneous targets
- Domain specific optimizations
OptiML Language Features

- 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]
    - Independent from the underlying implementation
  - Special data types: TrainingSet, TestSet, IndexVector, Image, Video ..
    - Encode semantic information

- Implicitly parallel control structures
  - sum{...}, (0::end) {...}, gradient { ... }, untilconverged { ... }
  - Allow anonymous functions with restricted semantics to be passed as arguments of the control structures
Example OptiML / MATLAB code (Gaussian Discriminant Analysis)

OptiML code

```matlab
val sigma = sum(0,x.numSamples) {
    if (x.labels(_ == false) {
        (x(_)-mu0).trans.outer(x(_)-mu0)
    } else {
        (x(_)-mu1).trans.outer(x(_)-mu1)
    }
}
```

MATLAB code (parallel)

```matlab
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
```
Benchmark Applications

- 6 machine learning applications
  - Gaussian Discriminant Analysis (GDA)
    - Generative learning algorithm for probability distribution
  - Loopy Belief Propagation (LBP)
    - Graph based inference algorithm
  - Naïve Bayes (NB)
    - Supervised learning algorithm for classification
  - K-means Clustering (K-means)
    - Unsupervised learning algorithm for clustering
  - Support Vector Machine (SVM)
    - Optimal margin classifier using SMO algorithm
  - Restricted Boltzmann Machine (RBM)
    - Stochastic recurrent neural network
Experimental Setup

- 4 Different implementations
  - OptiML+Delite
  - MATLAB (Original, GPU, Jacket)

- System 1: Performance Tests
  - Intel Nehalem
  - 2 sockets, 8 cores, 16 threads
  - 24 GB DRAM
  - NVIDIA GTX 275 GPU

- System 2: Scalability Tests
  - Sun Niagara T2+
  - 4 sockets, 32 cores, 256 threads
  - 128 GB DRAM
OptiML vs. MATLAB

GDA

Naive Bayes

Linear Regression

K-means

RBM

SVM

OptiML  MATLAB  Jacket
Measuring Intracellular Signaling with Mass Cytometry

- Bioinformatics Algorithm
  - Spanning-tree Progression Analysis of Density-normalized Events (SPADE)
  - P. Qiu, E. Simonds, M. Linderman, P. Nolan
SPADE is computationally intensive

Processing time for 30 files:

Matlab (parfor & vectorized loops)
2.5 days

C++ (hand-optimized OpenMP)
2.5 hours

…what happens when we have 1,000 files?
**SPADE Downsample: OptiML**

B. Wang and A. Sujeeth

```scala
for(node <- G.nodes if node.density == 0) {
  val (closeNbrs, closerNbrs) =
    node.neighbors filter {dist(_, node) < kernelWidth} 
    {dist(_, node) < approxWidth}
  node.density = closeNbrs.count
  for(nbr <- closerNbrs) {
    nbr.density = closeNbrs.count
  }
}
```

Downsample:

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

B. Wang and A. Sujeeth
while sum(local_density==0)~=0
    % process no more than 1000 nodes each time
    ind = find(local_density==0); ind = ind(1:min(1000,end));

    data_tmp = data(:,ind);
    local_density_tmp = local_density(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

for i=1:size(data_tmp,2)
    local_density_tmp(i) = sum(all_dist(i,:) < kernel_width);
    local_density(all_dist(i,:) < apprx_width) = local_density_tmp(i);
end
OptiML vs. C++

- OptiML provides much simpler programming model
- OptiML performance as good as C++ on full applications
More DSLs ...

- Graphs
  - Social networks, data analysis

- Visualization
  - Data analysis

- Query Language
  - Data analysis, financial trading

- Bio-simulation
  - Molecular dynamics, cells & viruses, drug-design, prosthetics

- Bio-informatics
  - Data analysis

- Your DSL goes here
New Problem

- We need to develop all of these DSLs

- Current DSL methods are unsatisfactory
Current DSL Development Approaches

- **Stand-alone DSLs**
  - Can include extensive optimizations
  - Enormous effort to develop to a sufficient degree of maturity
    - Actual Compiler/Optimizations
    - Tooling (IDE, Debuggers,...)
  - Interoperation between multiple DSLs is very difficult

- **Purely embedded DSLs ⇒ “just a library”**
  - Easy to develop (can reuse full host language)
  - Easier to learn DSL
  - Can Combine multiple DSLs in one program
  - Can Share DSL infrastructure among several DSLs
  - Hard to optimize using domain knowledge
  - Target same architecture as host language

Need to do better
Need to Do Better

- Goal: Develop embedded DSLs that perform as well as stand-alone ones.

- Intuition: General-purpose languages should be designed with DSL embedding in mind.
DSL Embedding Language

- Mixes OO and FP paradigms
  - Targets JVM
- Stanford/EPFL collaboration on leveraging Scala for parallelism
- “Language Virtualization for Heterogeneous Parallel Computing” Onward 2010, Reno
What is Scala

- Statically typed
- Allows OOP
  - Improves it by adding traits
  - Everything is an object
- Allows FP
  - Functions are just like any other object
    - can be passed to and returned from other functions/methods
  - Also offers closures
- Compiles to JVM/.NET
- Succinct, elegant and flexible syntax
  - Type inference
  - Syntactic sugaring features make libraries feel “native”
- Sophisticated type system
- Scalable language
Lightweight Modular Staging Approach

Typical Compiler

GPCE’10: Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs
**Delite: A Framework for DSL Parallelism**

H. Chafi, A. Sujeeth, K. Brown, H. Lee

DSLs adopt front-end from highly expressive embedding language but can customize IR and participate in backend phases

Need a framework to simplify development of DSL backends
Benefits of a Shared DSL Infrastructure

- Parallelization of DSLs
  - Map domain abstractions to low level parallel execution patterns
    - Why not provide a collection of these execution patterns?

- Code Generation to multiple targets
  - Most DSLs have common functionality
    - Why no handle this once and for all DSLs?

- Shared parallel runtime to accelerate DSL development

- Handle other issues in a consistent fashion across DSLs
  - Debugging
  - IDE and tooling support
Delite DSL Compiler

- Provide a common IR that can be extended while still benefitting from generic analysis and opt.
- Extend common IR and provide IR nodes that encode data parallel execution patterns
  - Now can do parallel optimizations and mapping
- DSL extends appropriate data parallel nodes for their operations
  - Now can do domain-specific analysis and opt.
- Generate an execution graph, kernels and data structures
The Delite IR

Application

M1 = M2 + M3
V1 = exp(V2)
s = sum(M)
C2 = sort(C1)

DS IR

Matrix Plus
Vector Exp
Matrix Sum
Collection Quicksort

Delite Op IR

ZipWith
Map
Reduce
Divide & Conquer

Base IR

Expression

Domain User Interface

Domain Analysis & Opt.

Parallelism Analysis & Opt.

Code Generation

Generic Analysis & Opt.

DSL User

DSL Author

Delite

Delite
Delite Execution

- Maps the machine-agnostic DSL compiler output onto the machine configuration for execution
- Walk-time scheduling produces partial schedules
- Code generation produces fused, specialized kernels to be launched on each resource
- Run-time executor controls and optimizes execution
Conclusions

- DSLs have potential to solve the heterogeneous parallel programming problem
  - Don’t expose programmers to explicit parallelism unless they ask for it
- Look at the process of developing DSLs for parallelism
  - Need programming languages to be designed for flexible embedding (Scala)
  - Lightweight modular staging in Scala allows for more powerful embedded DSLs
  - Delite provides a framework for adding parallelism
- Explore DSLs and DSL infrastructure in this course