Taming Heterogeneous Parallelism with Domain Specific Languages

Kunle Olukotun
Pervasive Parallelism Laboratory
Stanford University
ppl.stanford.edu
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
Computing System Power

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

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
Heterogeneous Parallel Programming

- Pthreads
- OpenMP
- CUDA
- OpenCL
- Verilog
- VHDL
- MPI
- PGAS
- Sun T2
- Nvidia Fermi
- Altera FPGA
- Cray Jaguar
Too many different programming models
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
- Sun T2
- CUDA
- OpenCL
- Nvidia Fermi
- Verilog
- VHDL
- Altera FPGA
- MPI
- PGAS
- Cray Jaguar

Pervasive Parallelism Laboratory
The Ideal Parallel Programming Language

Performance

Productivity

Generality
Successful Languages

Performance

Productivity

Generality

C/C++

Java

Python

Ruby
True Hypothesis $\Rightarrow$ Domain Specific Languages

Performance
(Heterogeneous Parallelism)

Domain Specific Languages

Productivity

Generality
Domain Specific Languages

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

OpenGL®
MATLAB®
MySQL®
RiALS
TEX
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
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)

DSL Infrastructure
- Domain Embedding Language (Scala)
  - Polymorphic Embedding
  - Staging
  - Static Domain Specific Opt.
- Parallel Runtime (Delite)
  - Task & Data Parallelism
  - Locality Aware Scheduling

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

// x : TrainingSet[Double]
// mu0, mu1 : Vector[Double]

val sigma = sum0,x.numSamples) {
  if (x.labels(_) == false) {
    (x(_)-mu0).trans.outer(x(_)-mu0)
  }
  else {
    (x(_)-mu1).trans.outer(x(_)-mu1)
  }
}

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

% x : Matrix, y: Vector
% mu0, mu1: Vector

OptiML code

(parallel) MATLAB code

ML-specific data types

Implicitly parallel control structures

Restricted index semantics

ML-specific data types
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
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?
for(node <- G.nodes if node.density == 0) {
    val (closeNbrs,closerNbrs) =
        node.neighbors.filter {dist(_,node) < kernelWidth}
            .filter {dist(_,node) < approxWidth}
    node.density = closeNbrs.count
    for(nbr <- closerNbrs) {
        nbr.density = closeNbrs.count
    }
}
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
end
OptiML vs. C++

- OptiML provides much simpler programming model
- OptiML performance as good as C++ on full applications
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
Goal: Develop embedded DSLs that perform as well as stand-alone ones

Intuition: General-purpose languages should be designed with DSL embedding in mind
Mixes OO and FP paradigms
- Targets JVM

Expressive type system allows powerful abstraction

Scalable language

Stanford/EPFL collaboration on leveraging Scala for parallelism

“Language Virtualization for Heterogeneous Parallel Computing” Onward 2010, Reno
Lightweight Modular Staging Approach

Modular Staging provides a hybrid approach

- DSLs adopt front-end from highly expressive embedding language
- Stand-alone DSL implements everything
- Can customize IR and participate in backend phases

Typical Compiler

LEXER → PARSER → TYPE CHECKER → ANALYSIS → OPTIMIZATION → CODE GEN

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
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

- **Domain User Interface**
  - Application
    - \( M_1 = M_2 + M_3 \)
    - \( V_1 = \exp(V_2) \)
    - \( s = \text{sum}(M) \)
    - \( C_2 = \text{sort}(C_1) \)

- **Domain Analysis & Opt.**
  - DS IR
    - Matrix Plus
    - Vector Exp
    - Matrix Sum
    - Collection Quicksort

- **Parallelism Analysis & Opt.**
  - Delite Op IR
    - ZipWith
    - Map
    - Reduce
    - Divide & Conquer

- **Code Generation & Execution**
  - Base IR
    - Expression

- **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
  - Determinism is a byproduct
- Need to simplify the process of developing DSLs for parallelism
  - Need programming languages to be designed for flexible embedding
  - Lightweight modular staging in Scala allows for more powerful embedded DSLs
  - Delite provides a framework for adding parallelism
- Early embedded DSL results are very promising