Composition and Reuse with Compiled Domain-Specific Languages

Pervasive Parallelism (PPL) Lab (http://ppl.stanford.edu) and LAMP (http://lamp.epfl.ch)

Arvind Sujeeth, Kevin Brown, HyoukJoong Lee, Hassan Chafi, Michael Wu, Victoria Popic, Kunle Olukotun
Tiark Rompf, Aleksander Prokopec, Vojin Jovanovic, Martin Odersky

Compiled Embedded Parallel DSLs

OptiLA: Linear Algebra
- •Dense/Sparse Vector/Matrix
- •Can be re-used & extended by other DSLs

OptiML: Machine Learning
- •Extends OptiLA
- •Adds ML-specific constructs: e.g., TrainingSet, untilconverged

OptiCollections: Parallel Collections
- •Arrays, Maps - Parallel collection interface using Delite ops
- •Designed to be used by other DSLs

OptiQL: Data Querying
- •In-memory collection-based querying
- •Utilizes OptiCollections' HashMap

OptiMesh: Mesh-based PDEs
- •Liszt on Delite
- •More advanced DSL analyses and optimizations

OptiGraph: Small-world Graphs
- •Green-Marl on Delite
- •Shares analyses / transformations with OptiMesh

DSL Re-Use

DSL	Delite Ops	Generic Optimizations
OptiML	Map, ZipWith, Reduce, Foreach, Hash, Sort	CSE, DCE, code motion, fusion
OptiQL	Map, Reduce, Filter, Sort, Hash, Join	CSE, DCE, code motion, fusion
OptiGraph	ForeachReduce, Map, Reduce, Filter	CSE, DCE, code motion
OptiMesh	ForeachReduce	CSE, DCE, code motion
OptiCollections	Map, FlatMap, Reduce, Filter, Sort, Hash, ZipWith	CSE, DCE, code motion, fusion

DSL Interoperability

We modified the scala-virtualized compiler to introduce *scopes*, coarse-grained execution blocks that are compiled, staged and executed independently

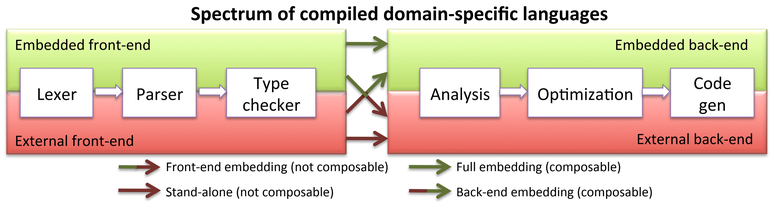
Scopes can communicate via global heap objects

```
OptiQL {
  // customers: Array[Customer]
  // orders: Array[Order]
  val orders = customers Join(orders)
    WhereEq(_.c_custkey, _.o_custkey)
    Select((c,o) => new Result {
      val nationKey = c.c_nationkey
      val acctBalance = c.c_acctbal
      val price = o.o_totalprice
    })
  orderData.set(orders)
}

OptiML {
  // run linear regression on price
  val data = Matrix(orderData.get)
  val x = data.sliceCols(0,1)
  val y = data.getCol(2)
  theta.set(linreg.weighted(x,y).toArray)
}

println("theta: " + theta.get)
```

Domain-Specific Languages for Heterogeneous Parallelism



Why Compiled Domain-Specific Languages?

- We want programs that satisfy the 3 P's:
  - Performance, Productivity, Portability
- Very hard to achieve with a general-purpose language: compilers lack semantic information necessary to efficiently translate to parallel hardware
- Domain-specific languages have been shown to be capable of providing the 3 P's in some domains, but:
  - *Embedded* DSLs are easy to build and compose, but don't perform well
  - *Stand-alone* DSLs are hard to build and compose, but perform well
- We propose using compiled DSLs embedded in a common *back-end* compiler

How do we compose them?

Two main ways:
- Open World: fine-grained cooperative composition
  - Embedded DSLs that are designed to be composed
  - No whole program analyses or optimizations that rely on domain assumptions or restricted semantics
  - DSL composition reduces to object composition in the host lang

```
trait Vectors extends MathOps with ScalarOps with VectorOps
trait Matrices extends Vectors with MatrixOps
```

Mix-in composition

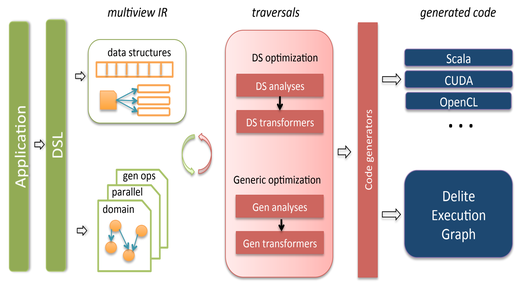
- Closed World: coarse-grained isolated composition
  - Embedded DSLs that require isolation to guarantee safety of optimizations
  - 3 steps: 1) independently parse DSL blocks and apply domain-specific optimizations, 2) lower each DSL block to a common, high-level IR, 3) optimize and code generate resulting IR

Delite and Lightweight Modular Staging

Lightweight Modular Staging: Embedded DSL compilation with metaprogramming

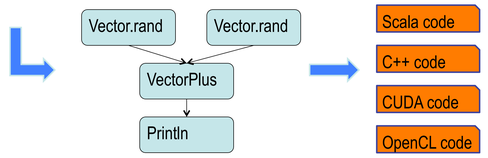
- All application types are wrapped in an abstract type constructor, Rep[T]
- The DSL returns **type-safe placeholders** instead of immediate results, allowing it to build an intermediate representation, optimize it, and generate code

Delite: Reusable infrastructure for rapid development of compiled embedded DSLs



Example DSL: Vectors

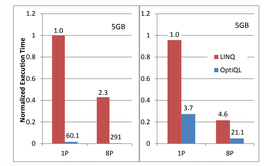
```
val (v1,v2) = (Vector.rand(1000), Vector.rand(1000))
val a = v1+v2
println(a)
```



- The DSL is embedded in **Scala**, a general purpose programming language
- The syntax is legal Scala
- When the program is executed, it builds an intermediate representation instead of computing a result **(DSL compilation)**
- The DSL implementation uses **Delite ops** to internally represent operations as parallel patterns:
  - `case class VectorPlus extends DeliteOpZipWith {`
    `def func = (a,b) => a + b`
    `}`
- Delite generates code from the DSL IR to multiple platforms (C++, Scala, CUDA, OpenCL)

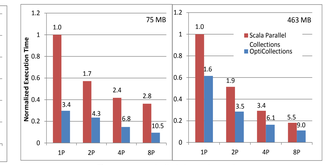
Evaluation

OptiQL vs. LINQ



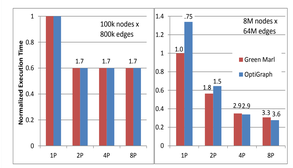
TPCH Q1 TPCH Q2

OptiCollections vs. Scala



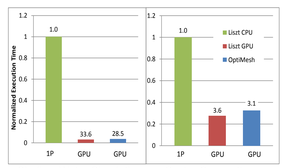
Reverse web-link benchmark

OptiGraph vs. Green Marl



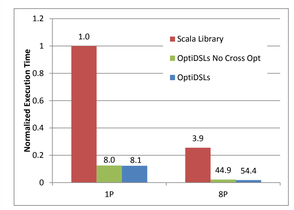
Page Rank

OptiMesh vs. Liszt

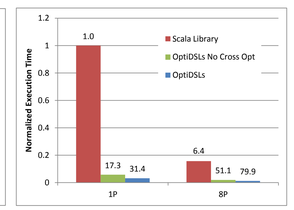


Shallow Water Simulation Scalar Convection

Open World Composition

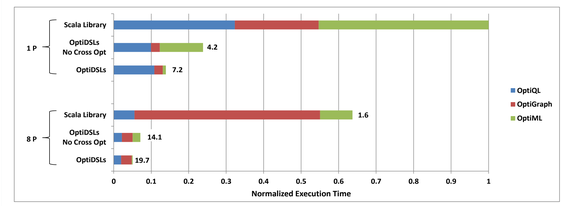


Small dataset



Large dataset

Closed World Composition



Twitter Data Analysis

Contact information

Open Source Repository
`http://stanford-ppl.github.com/Delite/`

Group Website
`http://ppl.stanford.edu`

E-mail Addresses (Students)
`asujeeth@stanford.edu,`

E-mail Address (Faculty Advisor)
`kunle@stanford.edu`