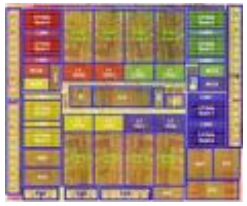
Delite: A Framework for Heterogeneous Parallel DSLs

Hassan Chafi, Arvind Sujeeth, Kevin Brown,
HyoukJoong Lee, Kunle Olukotun
Stanford University

Tiark Rompf, Martin Odersky
EPFL

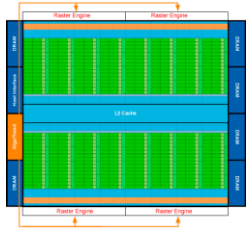
Heterogeneous Parallel Programming

Pthreads
OpenMP



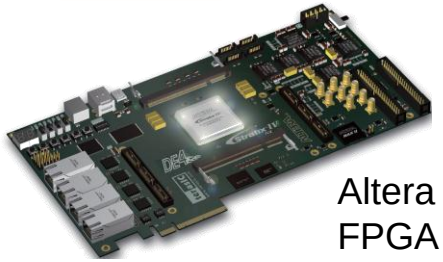
Sun
T2

CUDA
OpenCL



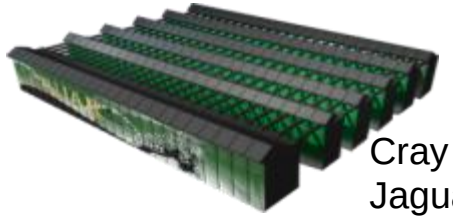
Nvidia
Fermi

Verilog
VHDL



Altera
FPGA

MPI



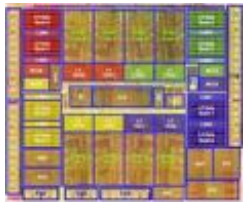
Cray
Jaguar

Programmability Chasm

Applications

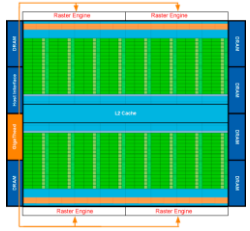
- Scientific Engineering
- Virtual Worlds
- Personal Robotics
- Data informatics

Pthreads
OpenMP



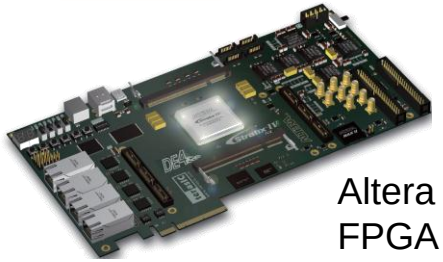
Sun
T2

CUDA
OpenCL



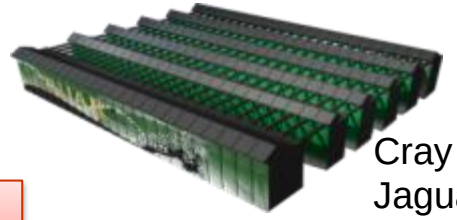
Nvidia
Fermi

Verilog
VHDL



Altera
FPGA

MPI



Cray
Jaguar

Too many different programming models

IS IT POSSIBLE TO WRITE ONE PROGRAM

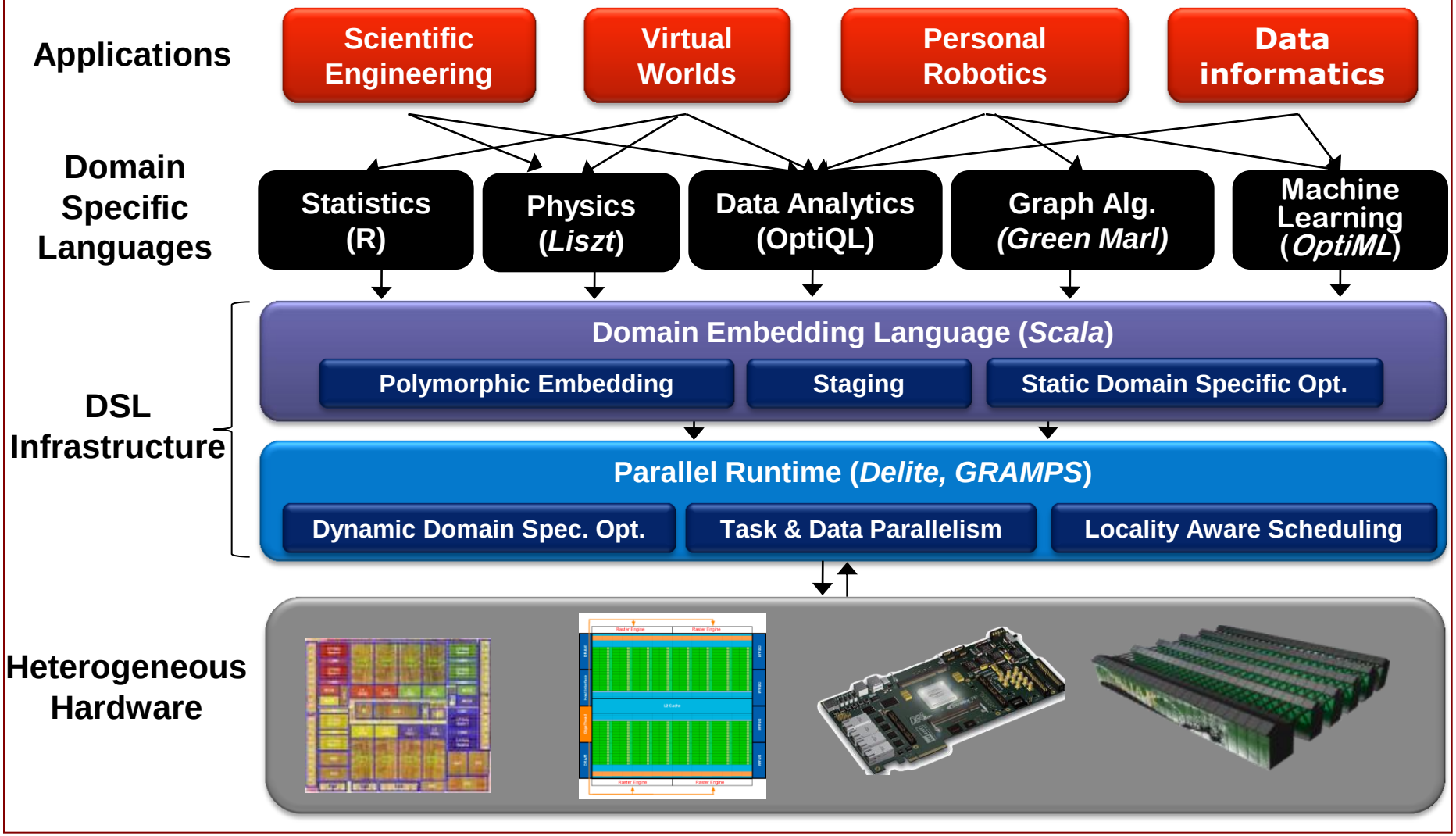
AND

RUN IT ON ALL THESE TARGETS?

HYPOTHESIS: YES, BUT NEED

DOMAIN-SPECIFIC LANGUAGES

PPL Vision


Applications
Scientific Engineering
Virtual Worlds
Personal Robotics
Data informatics
Domain Specific Languages
Statistics (R)
Physics (*Liszt*)
Data Analytics (OptiQL)
Graph Alg. *(Green Marl)*
Machine Learning (*OptiML*)
DSL Infrastructure
Domain Embedding Language (*Scala*)
Polymorphic Embedding
Staging
Static Domain Specific Opt.
Parallel Runtime (*Delite, GRAMPS*)
Dynamic Domain Spec. Opt.
Task & Data Parallelism
Locality Aware Scheduling
Heterogeneous Hardware

DSLs Present New Problem

We need to develop all 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

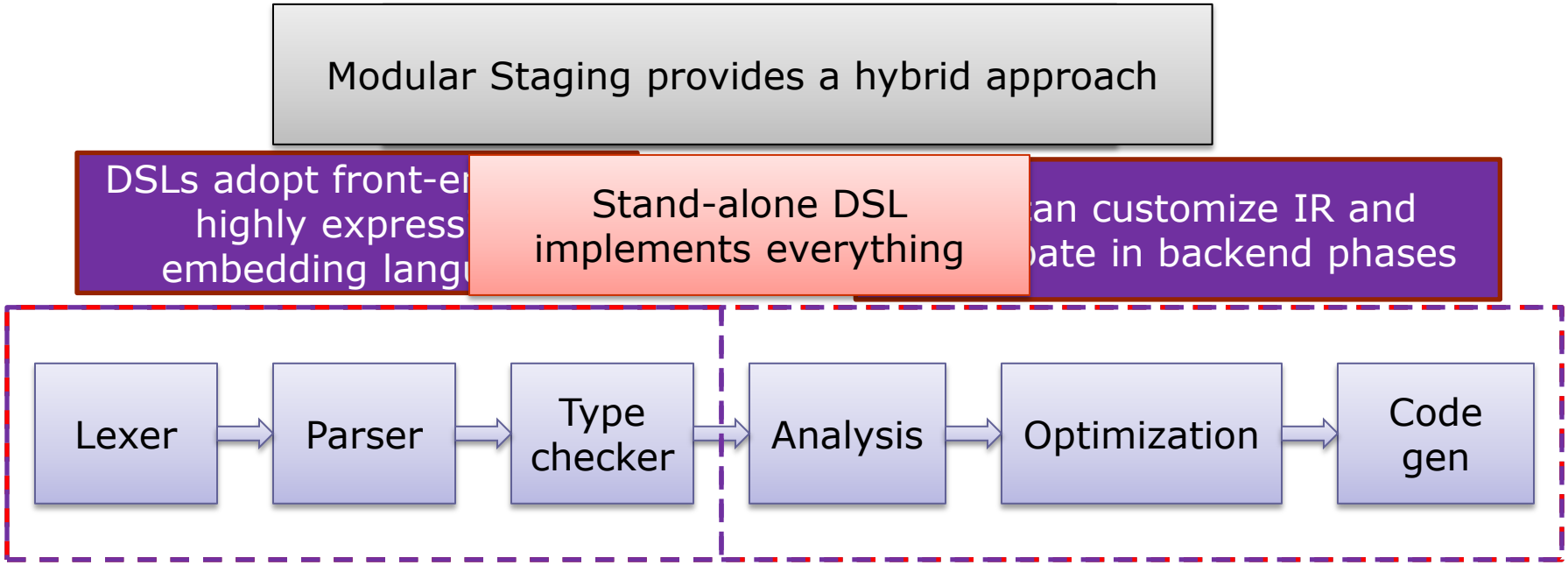
Need to do better

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

Modular Staging Approach



Typical Compiler

GPCE'10: Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs

Linear Algebra Example

```
object TestMatrix {


  def example(a: Matrix, b: Matrix, c: Matrix, d: Matrix) = {
    val x = a*b + a*c
    val y = a*c + a*d
    println(x+y)
  }

}
```

Abstract Matrix Usage

```
object TestMatrix extends DSLApp with MatrixArith {


  def example(a: Rep[Matrix], b: Rep[Matrix],
          c: Rep[Matrix], d: Rep[Matrix]) = {
    val x = a*b + a*c
    val y = a*c + a*d
    println(x+y)
  }

}
```

- Rep[Matrix]: abstract type constructor ⇒ range of possible implementations of Matrix
- Operations on Rep[Matrix] defined in MatrixArith trait

Lifting Matrix to Abstract Representation

- DSL interface building blocks structured as traits
  - Expressions of type Rep[T] *represent* expressions of type T
  - Can plug in different representation
- Need to be able to convert (lift) Matrix to abstract representation
- Need to define an interface for our DSL type

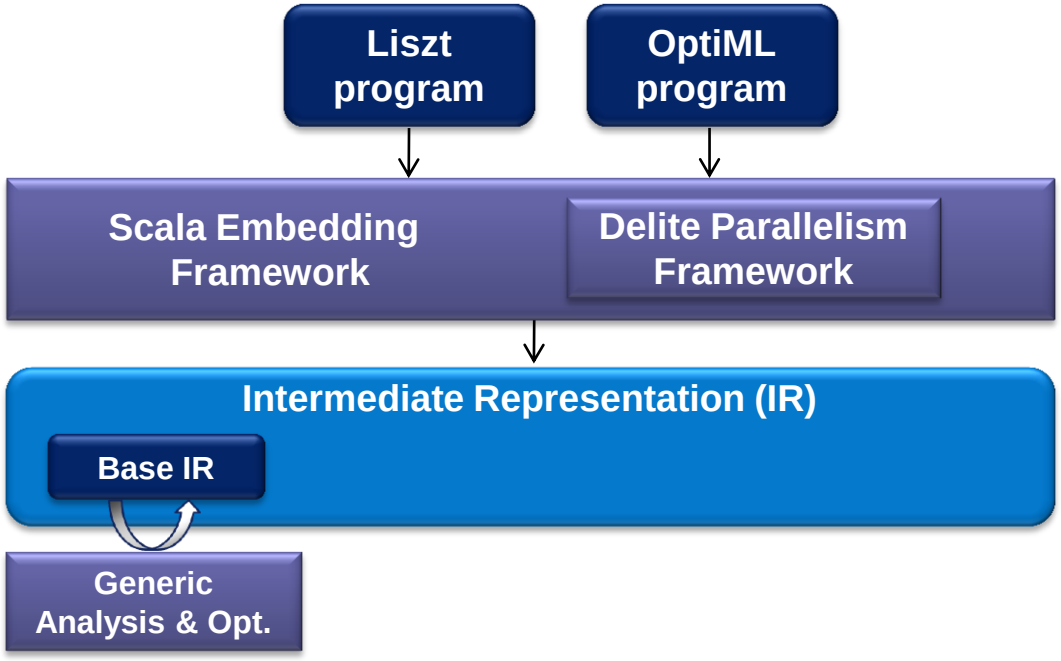
```
trait MatrixArith {

  type Rep[T]

  implicit def liftMatrixToRep(x: Matrix): Rep[Matrix]

  def infix_+(x:Rep[Matrix], y: Rep[Matrix]): Rep[Matrix]
  def infix_*(x:Rep[Matrix], y: Rep[Matrix]): Rep[Matrix]

}
```

- Now can plugin different implementations and representations for the DSL

Modest Effort

- Lifting each new DSL that uses slightly different IR violates Effort criterion
- Need a DSL embedding infrastructure
- Provide building blocks of common DSL compiler functionality

Delite DSL Compiler



- Provide a common IR that can be extended while still benefitting from generic analysis and opt.

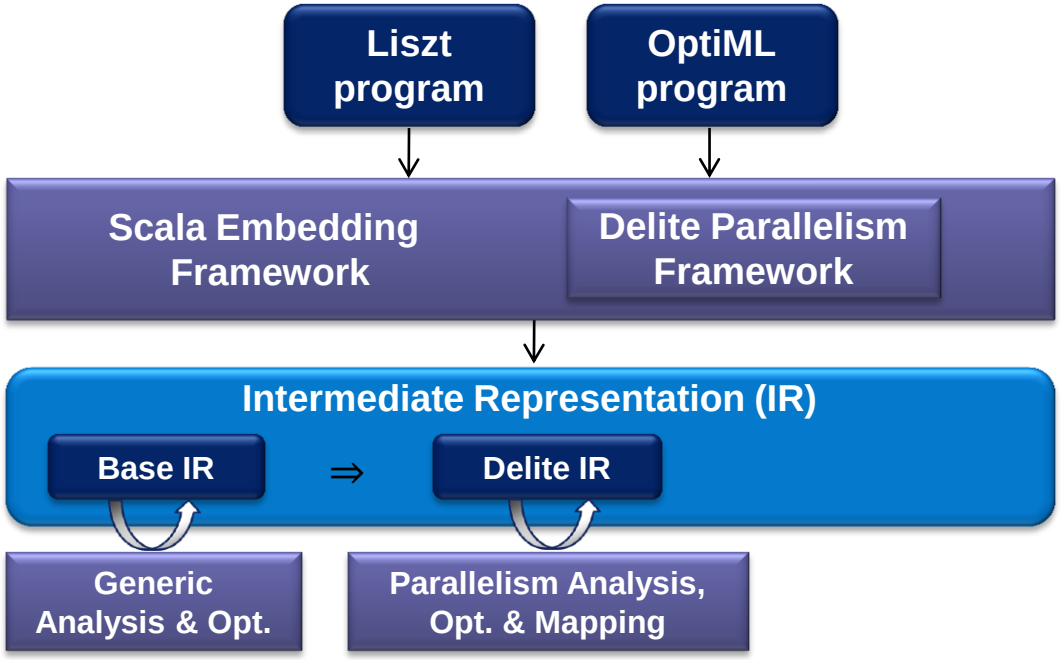
Now Can Build an IR

- Start with common IR structure to be shared among DSLs

```
trait Expressions {
     // constants/symbols (atomic)
    abstract class Exp[T]
    case class Const[T](x: T) extends Exp[T]
    case class Sym[T](n: Int) extends Exp[T]

    // operations (composite, defined in subtraits)
    abstract class Op[T]

    // additional members for managing encountered definitions
    def findOrCreateDefinition[T](op: Op[T]): Sym[T]

    implicit def toExp[T](d: Op[T]): Exp[T] = findOrCreateDefinition(d)
  }
```

- Generic optimizations provided to all DSLs
  - Common subexpression elimination
  - Dead code elimination
  - Code motion

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 parallel execution patterns
  - Now can do parallel optimizations and mapping

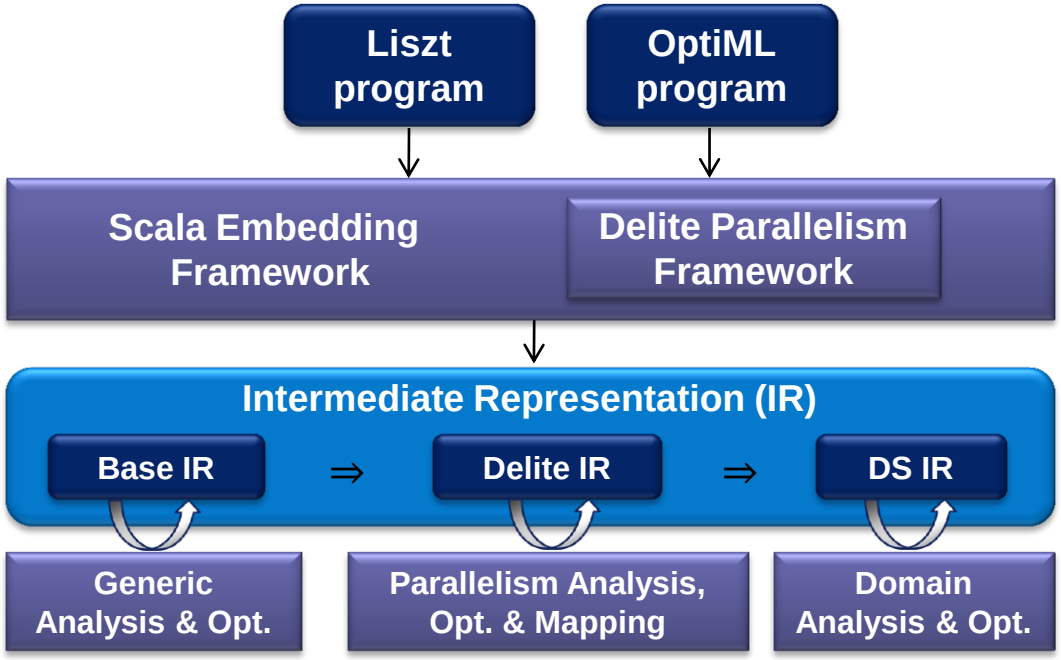
Delite Ops

- Encode known parallel execution patterns
  - Data-parallel: map, zip, reduce, …
- Delite provides implementations of these patterns for multiple hardware targets
  - e.g., multi-core, GPU
- DSL developer maps each domain operation to the appropriate pattern

Delite Op Fusing

- Operates on all loop-based ops
- Reduces op overhead and improves locality
  - Elimination of temporary data structures
  - Communication through registers
- Fuse both dependent and side-by-side operations
  - Fused ops can have multiple outputs
- Algorithm: fuse two loops if
  - size(loop1) == size(loop2)
  - No dependencies exist that would require an impossible schedule if fused
    - e.g., C depends on B, depends on A -> cannot fuse C and A

Delite DSL Compiler



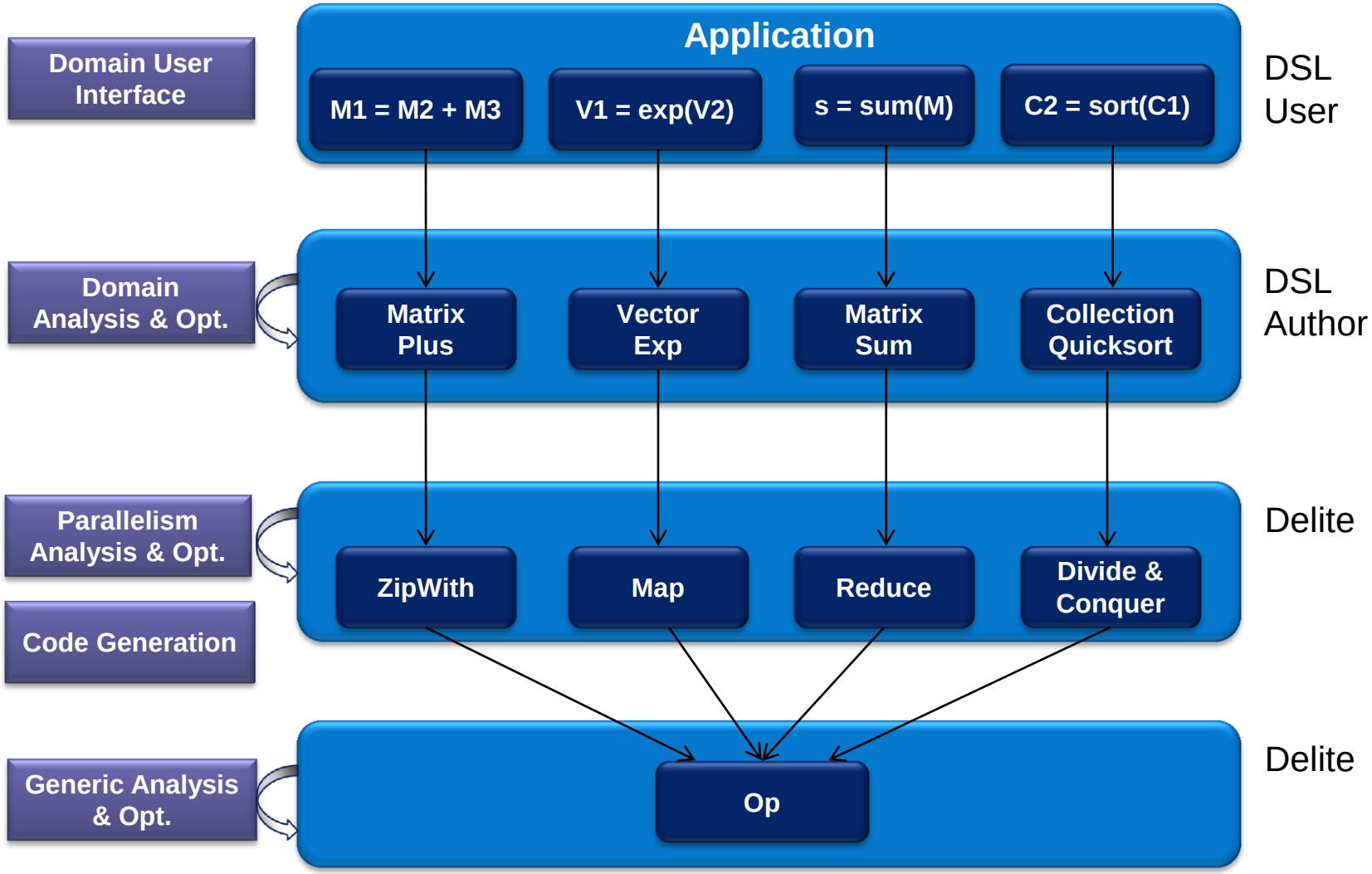
- Provide a common IR that can be extended while still benefitting from generic analysis and opt.
- Extend common IR and provide IR nodes that encode parallel execution patterns
  - Now can do parallel optimizations and mapping
- DSL extends appropriate parallel nodes for their operations
  - Now can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

Customize IR with Domain Info

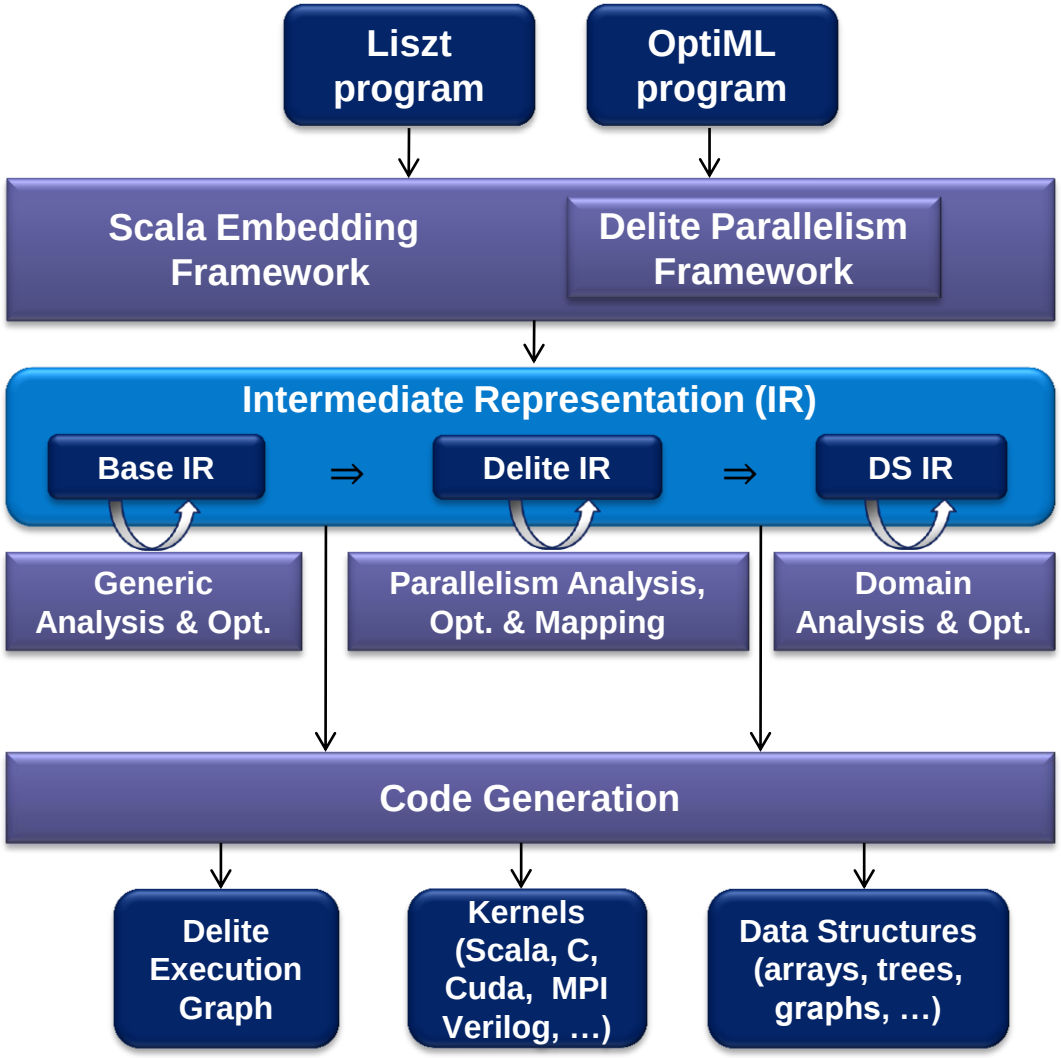
```
trait MatrixArithRepExp extends MatrixArith with Expressions {
  type Rep[T] = Exp[T]
  implicit def liftMatrixToRep(x: Matrix) = Const(x)
  case class Plus(x: Exp[Matrix],y: Exp[Matrix]) extends DeliteOpZip[Matrix]
  def infix_+(x: Exp[Matrix],y: Exp[Matrix]) = Plus(x, y)
}
```

- Choose Exp as representation for the DSL types
- Define Lifting function to create expressions
- Extend Delite IR with domain-specific node types
- DSL methods build IR as program runs

The Delite IR

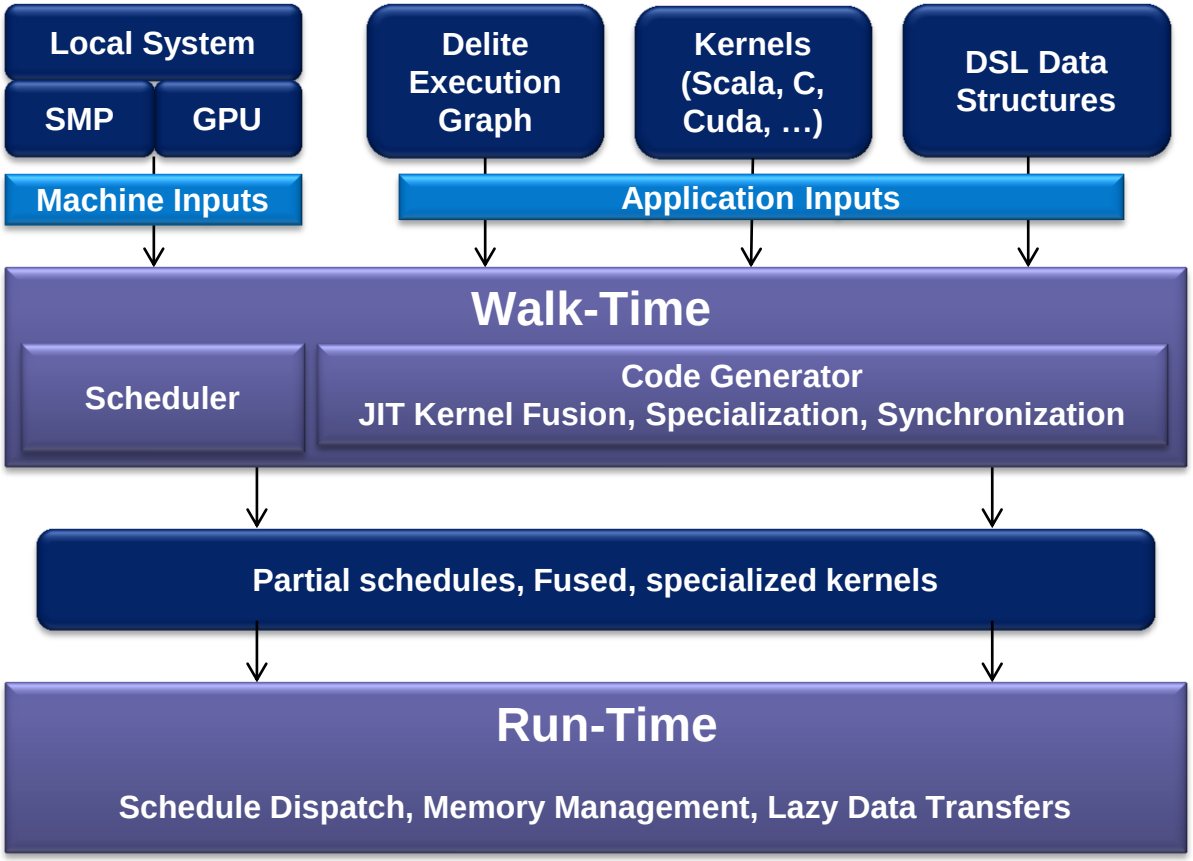


Delite DSL Compiler



- 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

Delite Runtime



- Maps the machine-agnostic DSL compiler output onto the given machine specification for customized execution
- Walk-time scheduling produces partial schedules
  - Generates variants to be selected at run-time once actual execution flow is resolved
- Code generation produces fused, specialized kernels to be launched on each resource
  - Optimizes for the execution plan using the partial schedules
- Run-time executor controls and optimizes execution
  - Launches kernels on resources
  - Monitors execution and system state to dynamically adjust kernels and schedule

Schedule and Kernel Compilation

- Compile execution graph to executables for each resource after scheduling
  - All synchronization is added at this point where required
- Kernels specialized based on number of processors allocated for it
  - e.g., specialize height of tree reduction

GPU Management

- Cuda host thread launches kernels and automatically performs data transfers as required by schedule
  - Compiler provides helper functions to
    - Copy data structures between address spaces
    - pre-allocate outputs and temporaries
    - select the number of threads & thread blocks

- Provides device memory management for kernels
  - Perform liveness analysis to determine when op inputs and outputs are dead on the GPU
  - Runtime frees dead data when it experiences memory pressure

Conclusions

- 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