Delite

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

Tiark Rompf, Martin Odersky
EPFL

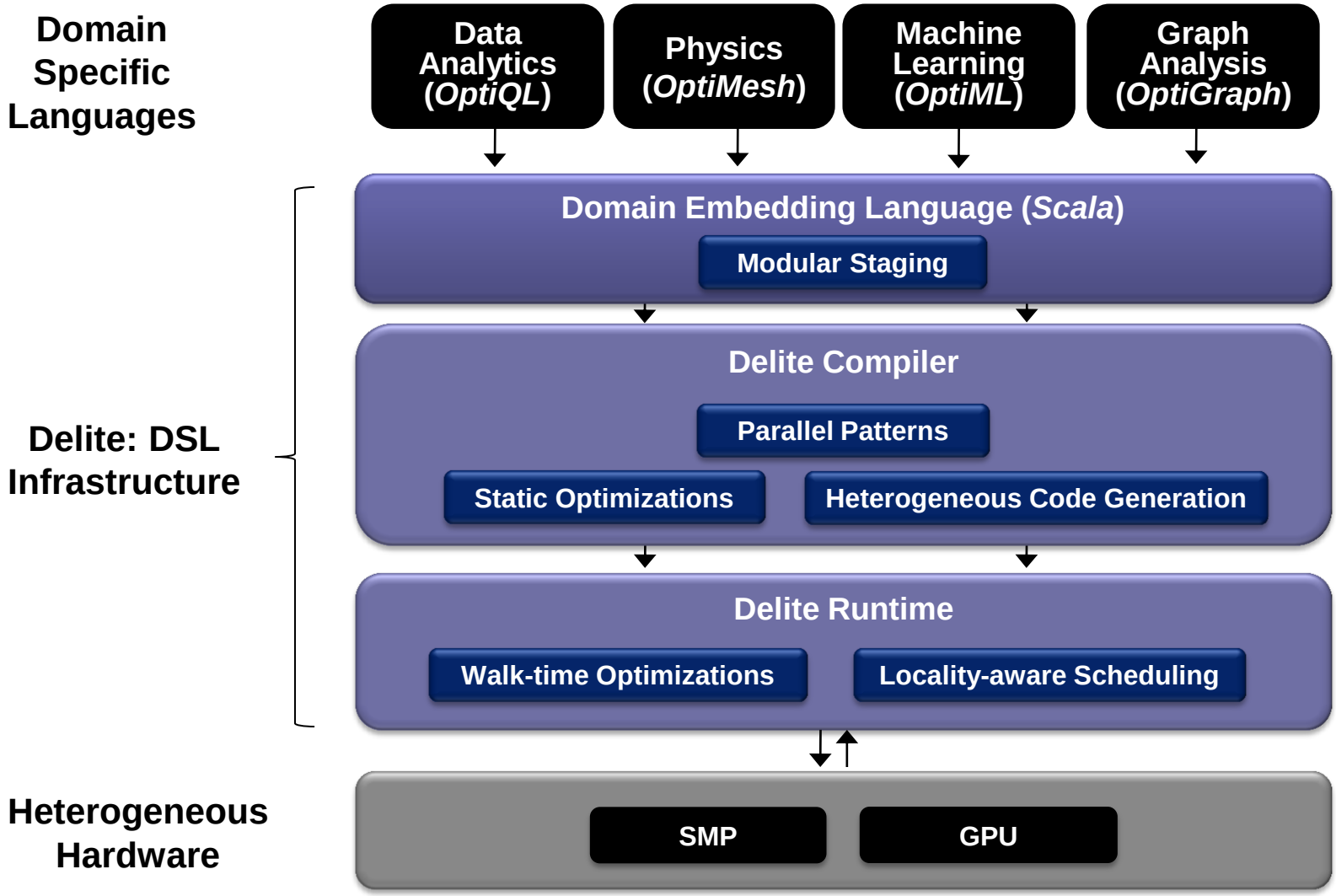
Building a Language is Difficult

- Building a new DSL
  - Design the language (syntax, operations, abstractions, etc.)
  - Implement compiler (parsing, type checking, optimizations, etc.)
  - Discover parallelism (understand parallel patterns)
  - Emit parallel code for different hardware (optimize for low-level architectural details)
  - Handle synchronization, multiple address spaces, etc.
- Need a DSL infrastructure
  - Embed DSLs in a common host language
  - Provide building blocks for common DSL compiler & runtime functionality

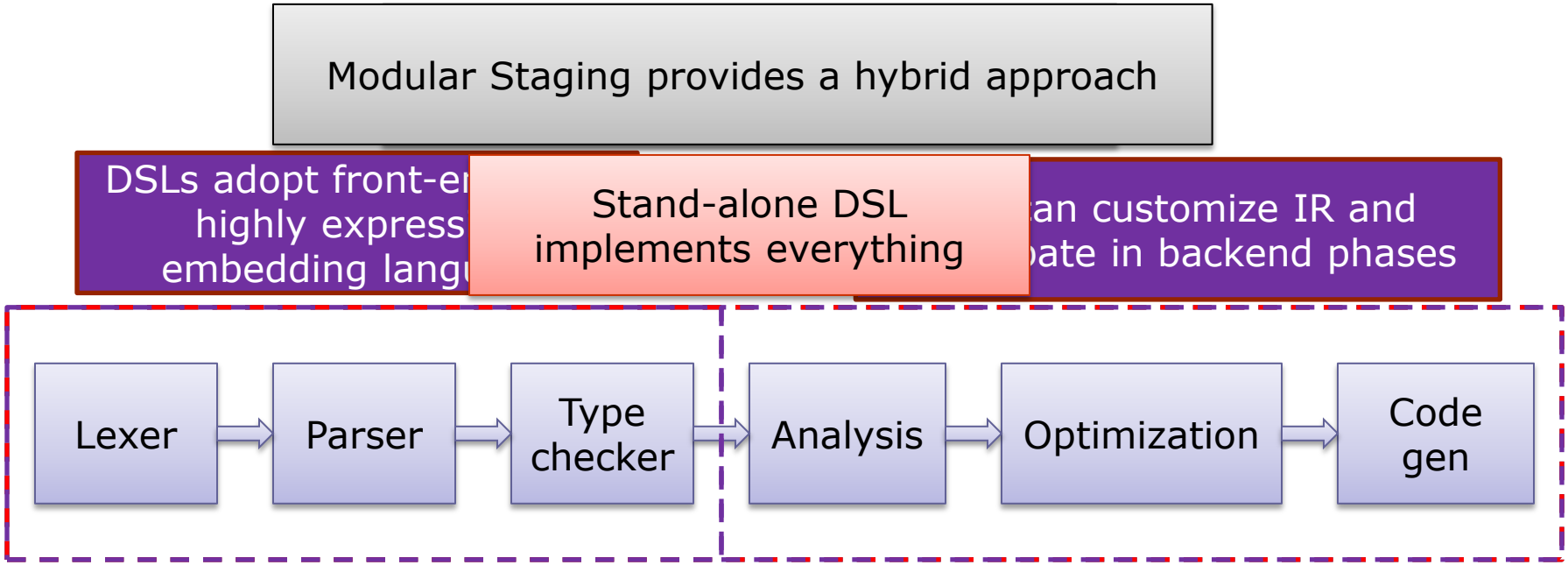
Delite Approach

- Embed DSLs in Scala to leverage Scala compiler's front-end (parsing, type-checking)
- Provide a reusable IR in order to share common optimizations across DSLs
  - Extend to add new optimizations
- Provide parallel patterns that we implement on heterogeneous architectures efficiently

Delite Overview



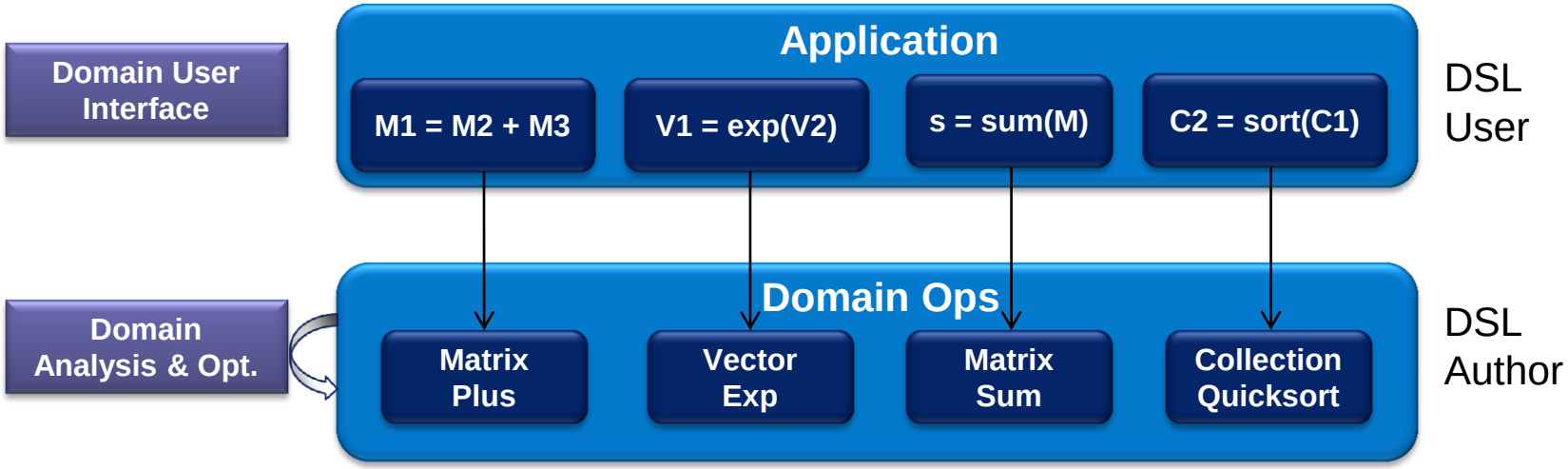
Modular Staging Approach



Typical Compiler

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

DSL Intermediate Representation (IR)



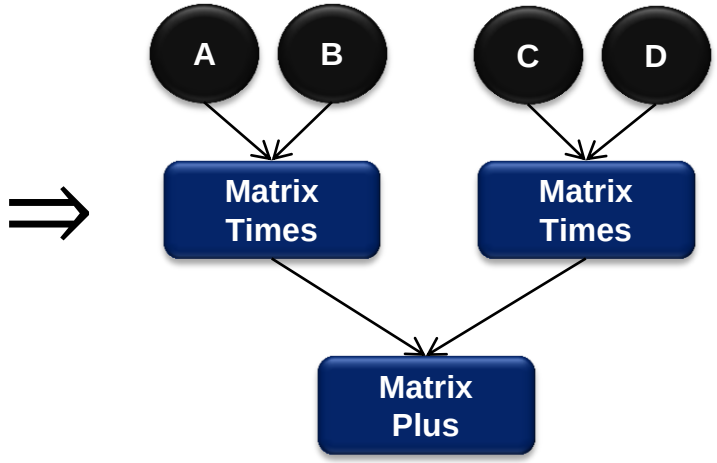
Matrix Example

```
import Matrix


trait 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)
  }
}
```

- How do we construct an IR of this?

Building an IR

```
def infix_+(a: Matrix, b: Matrix) =

def infix_*(a: Matrix, b: Matrix) =
```



- DSL methods build IR as program runs

Abstract Matrix Representation

```
trait TestMatrix extends 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 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) base types to abstract representation
- Need to define an interface for our DSL type

```
trait Base {
  type Rep[T]
  implicit def unit[T](x: T): Rep[T]
}

trait MatrixArith extends Base {
  def infix_+(a:Rep[Matrix], b: Rep[Matrix]): Rep[Matrix]
  def infix_*(a:Rep[Matrix], b: Rep[Matrix]): Rep[Matrix]

 }
```

- Now can plugin different implementations and representations for the DSL

Lifting Control Structures

- Constructs of the embedding language can be overriden by the DSL:

```
if (cond) thenBlock else elseBlock
```

maps to

```
__ifThenElse(cond, thenBlock, elseBlock)
```

- DSL developer can control the meaning of conditionals by providing overloaded variants specialized to DSL types

Lifting Scala

- What we lift into the Rep context
  - DSL-defined operations
  - Basic Scala types (primitives, Arrays, Lists, Tuples, etc.)
  - Functions
  - Control structures (If, For, While, …)
  - Equality
  - Variable declaration and assignment

- What we don’t lift
  - Classes
  - Methods (will be evaluated at each call site)

Optimizing the IR

- DSL developer defines how DSL operations create IR nodes

```
def matrix_plus(x: Rep[Matrix], y: Rep[Matrix]) = MatrixPlus(x,y)
```

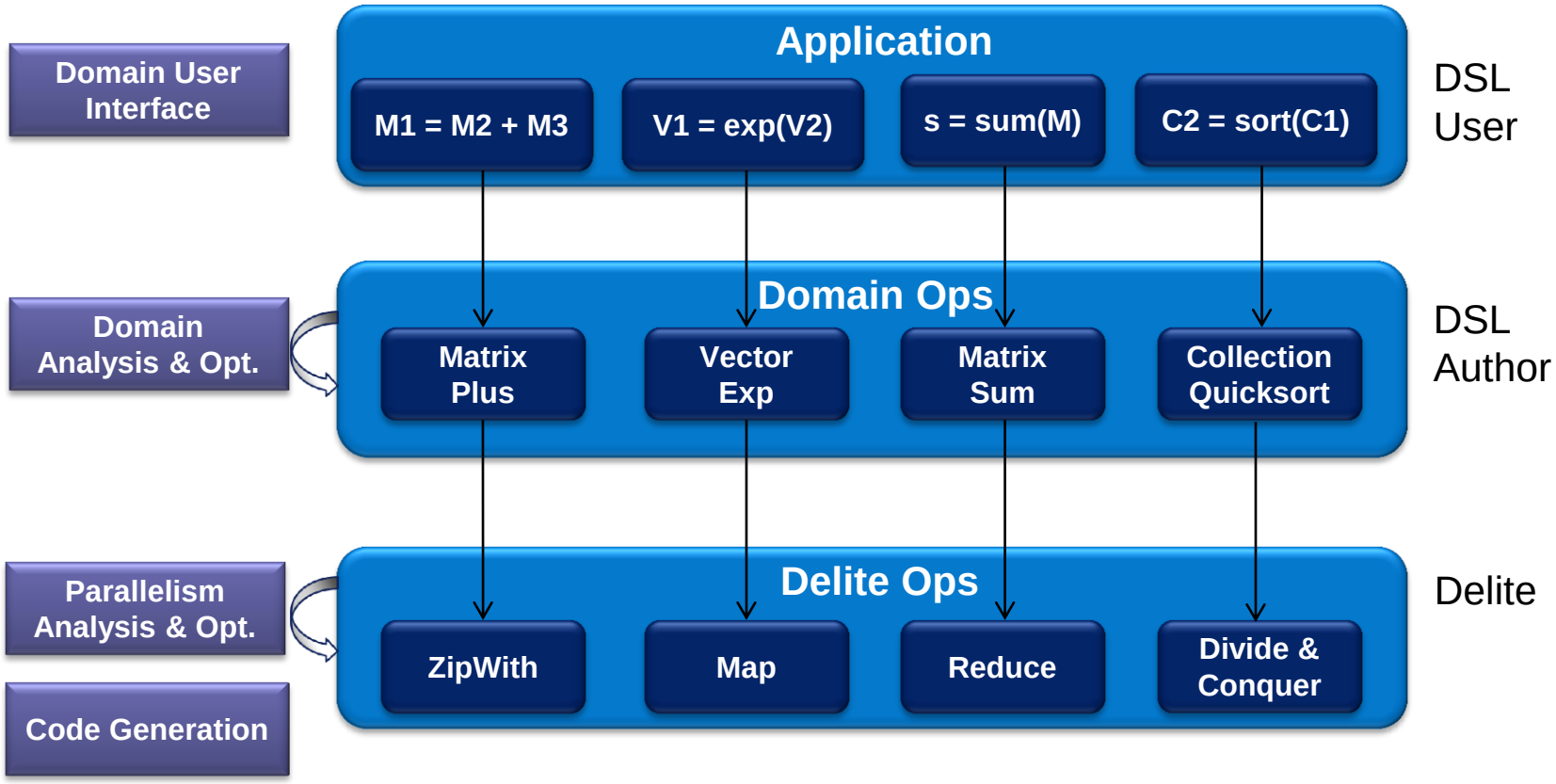
- Specialize implementation of operation for each *occurrence* by pattern matching on the IR

```
override def matrix_plus(x: Rep[Matrix], y: Rep[Matrix]) = (x, y) match {
  case (Def(MatrixTimes(a, b)), Def(MatrixTimes(c, d))) if (a == c) =>
    MatrixTimes(a, MatrixPlus(b,d))
  case _ => super.matrix_plus(x, y)
}
```

Delite DSL Framework

- Building a new DSL
  - Design the language (syntax, operations, abstractions, etc.)
  - Implement compiler
    - Domain-specific analysis and optimization
    - Lexing, parsing, type-checking, generic optimizations
  - Discover parallelism (understand parallel patterns)
  - Emit parallel code for different hardware (optimize for low-level architectural details)
  - Handle synchronization, multiple address spaces, etc.

Multiview Delite IR



Delite Ops

- Encode known parallel execution patterns
  - Map, Reduce, ZipWith, Foreach, Filter
  - Sequential
  - Hash (GroupBy), Join, Sort, ForeachReduce
- Delite provides implementations of these patterns for multiple hardware targets
  - e.g., multi-core, GPU
- DSL author maps each domain operation to the appropriate pattern
  - Delite handles parallel optimization, code generation, and execution for all DSLs

Using Delite Ops

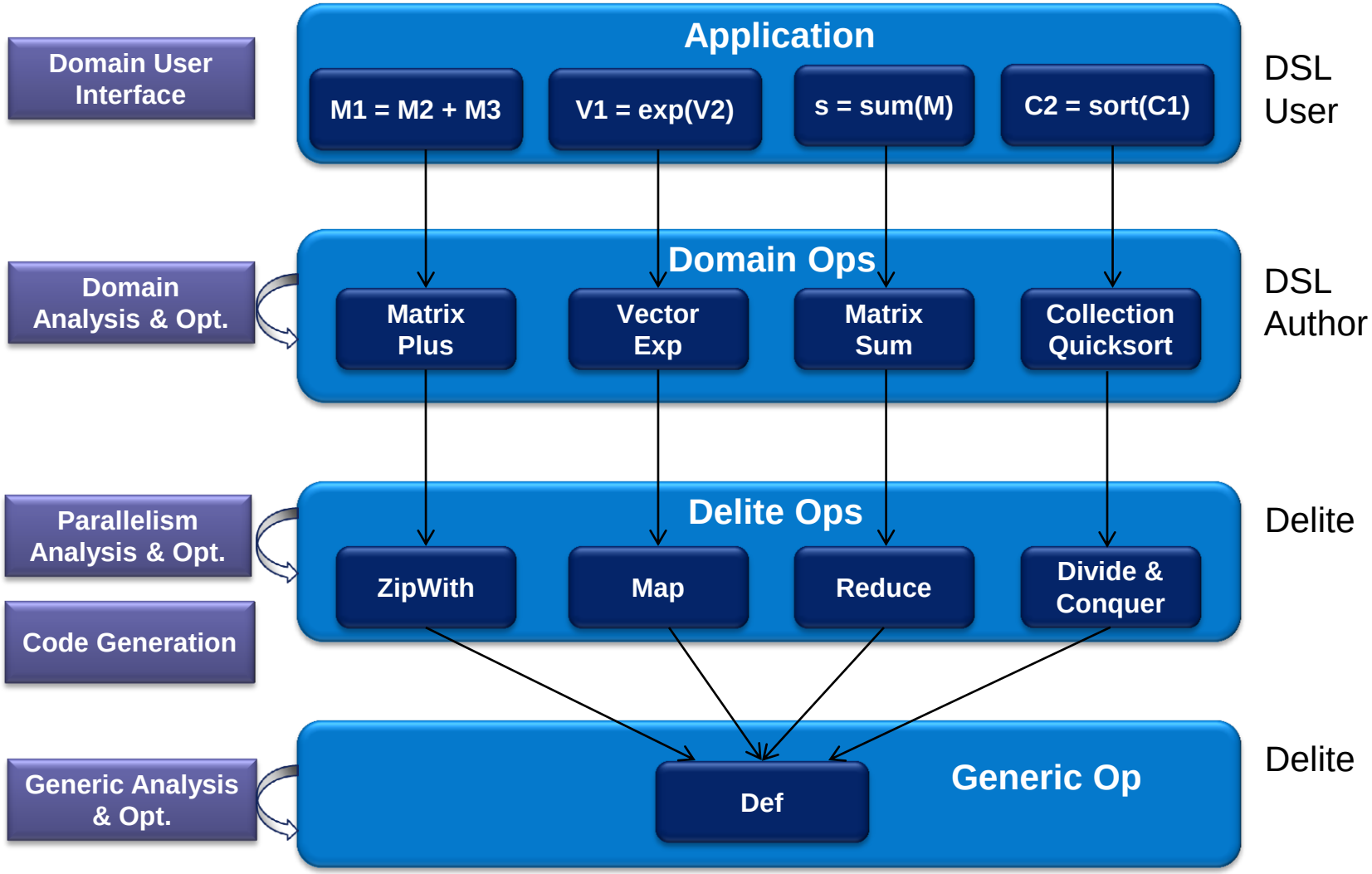
```
def vector_plus[A:Manifest:Numeric](x: Rep[Vector[A]], y: Rep[Vector[A]]) =
  VectorPlus(x,y)

case class VectorPlus[A:Manifest:Numeric](x: Rep[Vector[A]], y: Rep[Vector[A]])
  extends DeliteOpZipWith[A,A,Vector[A]] {

}
```

```
def vector_sum[A:Manifest:Numeric](x: Rep[Vector[A]]) =
  VectorSum(x)

case class VectorSum[A:Manifest:Numeric](x: Rep[Vector[A]])
  extends DeliteOpReduce[A] {

}
```

Delite Op Fusion

- Operates on DeliteOpMultiLoop
  - A common ancestor of all loop-based Delite Ops
- Reduces op overhead and improves locality
  - Elimination of temporary data structures
  - Merging loop bodies may enable further optimizations
- Fuse both producer-consumer and sibling operations
  - Fused ops can have multiple inputs & outputs
- Algorithm: can fuse if no cyclical dependencies &
  - Same input size (siblings), e.g., map;zip;reduce
  - consume map-like, e.g., map-zip-reduce, etc.
  - consume filter-like
  - hash-map{reduce} -> hashReduce

Multiview Delite IR



Common IR: Sea-of-Nodes

- No explicit control-flow graph
- Sea-of-node representation
  - Nodes only connected by data dependencies
  - Control dependencies (caused by effects) represented as additional data dependencies
  - Otherwise free to float anywhere – good for parallelism
- The main application method returns the result IR node
  - The entire program is then emitted via the compiler by scheduling the dependencies needed to emit that result
- Generic optimizations performed at the granularity of domain operations

Common IR Representation

```
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 Def[T]

  case class TP[T](lhs: Sym[T], rhs: Def[T])

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

  implicit def toAtom[T](d: Def[T]): Exp[T] = findOrCreateDefinition(d).sym
}
```

```
trait BaseExp extends Base {
  type Rep[T] = Exp[T]
  implicit def unit[T](x: T) = Const(x)
}
```

Generic Optimizations

- Common subexpression elimination
  - Global dictionary tracks what's been seen before

```
trait Expressions {
  implicit def toAtom[T](d: Def[T]): Exp[T] = findOrCreateDefinition(d).sym
}

trait MatrixOps {
  def infix_plus(a: Exp[Matrix], b: Exp[Matrix]): Exp[Matrix] = MatrixPlus(x,y)
  case class MatrixPlus(a: Exp[Matrix], b: Exp[Matrix]) extends DeliteOpZip
}
```

- Dead code elimination
  - All code is emitted via scheduling the dependencies of computing some required result
  - Dead code is never encountered in this process

Generic Optimizations (2)

- Constant folding
  - Operations on constants are computed as program stages
  - Constants are lifted into the IR lazily

```
trait MatrixApp {
  def example(a: Rep[Matrix[Int]]) {
    val x = 5
    val y = x + 3
    val ax = a * x
    val ay = a + y
  }
}
```

- Code motion
  - Pull computation out of loops and push computation into conditionals during code scheduling
  - Use a “hot/cold” heuristic to deal with nested loops/conditionals

Side Effects

- Reflection and reification model
  - An effect operation must be *reflected* at the point where its effect should occur
    - `def print(x: Exp[String]) = reflectEffect(Print(x))`
  - Reifying the effects of a block of code amounts to executing the code with an empty current state, and returning a representation of the result value together with the resulting state

```
def main() = {
  print("A")
  print("B")
  3+4
}
```

```
reifyEffects(main()) = Reify(Const(7), List(Print(Const("A")), Print(Const("B"))))
```

The Effects API

- DSL developer explicitly designates effectful operations with the following methods:
  - `reflectEffect[A](d: Def[A]): Exp[A]`

    `//a generic effect, e.g, print`
  - `reflectMutable[A](d: Def[A]): Exp[A]`

    `//instantiating a mutable object`
  - `reflectWrite[A](write: Exp[Any]*)(d: Def[A]): Exp[A]`

    `//d writes to the symbols in write`
  - `reflectPure[A](d: Def[A]): Exp[A]`

    `// a non-effectful operation, automatically converted to a read effect if it depends on something mutable`
- Serialize reads of anything that may alias one or more mutable objects with writes to those objects
- We do not allow nested mutable objects
  - Variables count as mutable

DSL Extensibility

- DSL extensibility
  - New DSLs achieve re-use by extending Delite provided constructs
  - New DSLs can also extend other existing DSLs
    - e.g., common Linear Algebra DSL at the core of other DSLs (machine learning, convex optimization, …)
- Mixins: traits select exactly what to include in your DSL
  - Delite provides basic Scala ops (PrimitiveOps, TupleOps, etc.) for any DSL
  - OptiCollections provides lifted mirror of the Scala collections library (Map, Set, etc.)
    - Operations implemented with DeliteOps
- Challenges:
  - Namespace conflicts: all traits mixed into single object
  - Combining optimizations & transformations: must be aware of semantics being composed

DSL Interoperability

- Application may not fit perfectly in a single DSL
  - Could require multiple DSLs (multiple application phases)
  - Fall back to general-purpose language for non performance-critical tasks
- Solution: create DSL "scopes" within Scala
  - A scope is a block of code that is compiled, staged, and executed as a chunk
    - Provides encapsulation for the DSL
  - Scopes can communicate with one another through global objects (shared address space)

Interoperability Example

```
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)
```

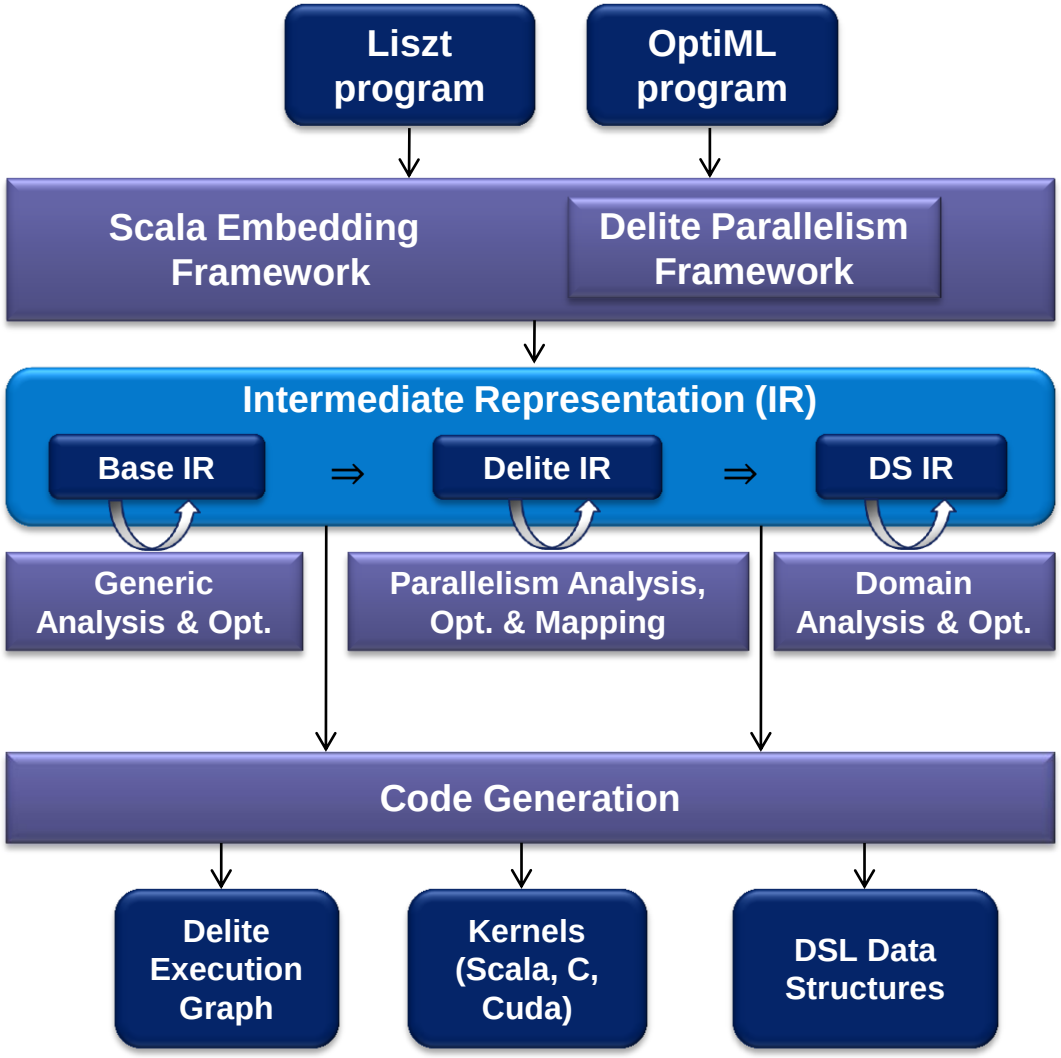
OptiQL DSL trait implicitly injected

Communicate through global object

OptiML DSL trait implicitly injected

standard Scala

Delite DSL Compiler Infrastructure



Heterogeneous Code Generation

- Delite can have multiple registered target code generators (Scala, Cuda, …)
  - Calls all generators for each Op to create kernels
  - Only 1 generator has to succeed
- Generates an *execution graph* that enumerates all Delite Ops in the program
  - Encodes parallelism within the application
  - Contains all the information the Delite Runtime requires to execute the program
    - Op dependencies, supported targets, etc.

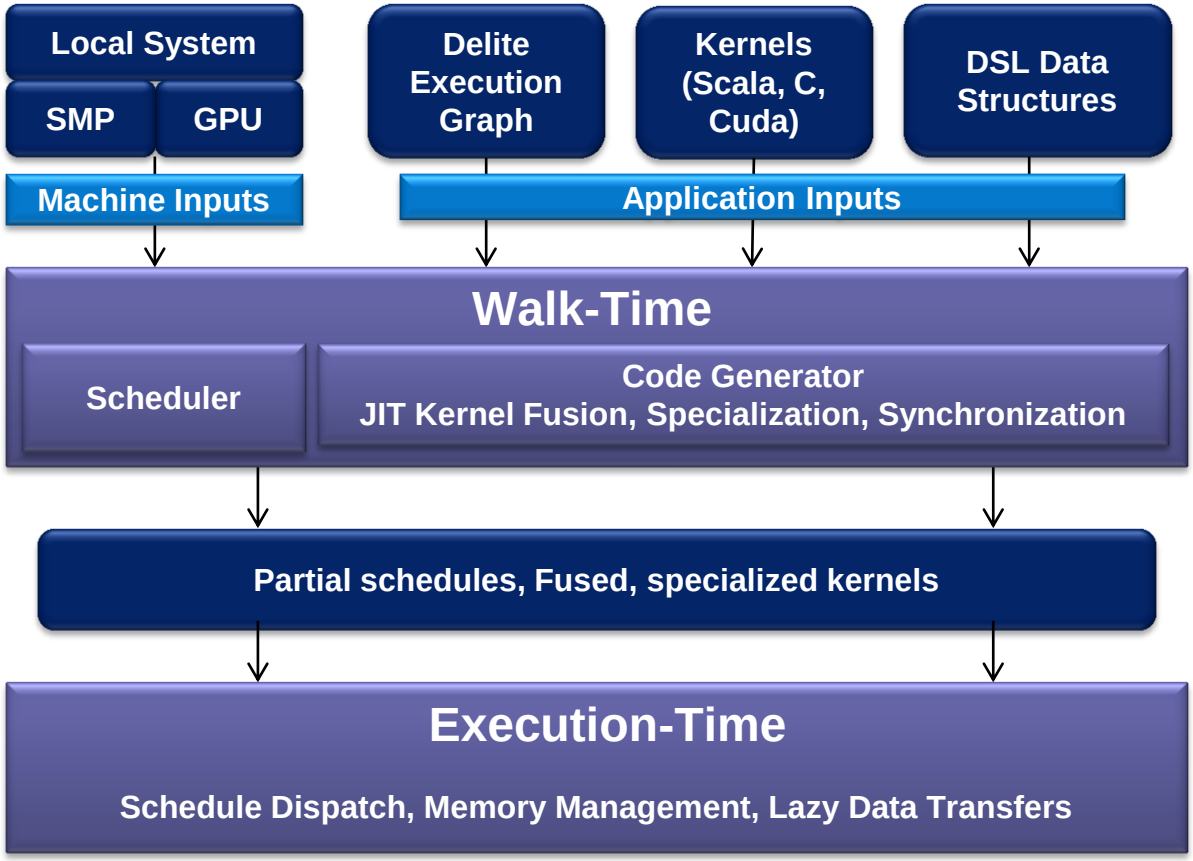
Code Generation

```
trait GenericCodegen
  def emitNode(sym: Sym[Any], rhs: Def[Any]) {
    throw new GenerationFailedException("don't know how to generate " + rhs)
}
```

```
trait ScalaGenMatrixOps extends ScalaGenBase
  override def emitNode(sym: Sym[Any], rhs: Def[Any]) = rhs match {
    case m@MatrixObjectNew(numRows, numCols) => emitValDef(sym, "new %s(%s,%s)".
         format(remap(m.mM), quote(numRows), quote(numCols)))

    case MatrixNumRows(x) => emitValDef(sym, quote(x) + ".numRows")
    //...
    case _ => super.emitNode(sym, rhs)
  }
}
```

```
trait CudaGenMatrixOps extends CudaGenBase
  override def emitNode(sym: Sym[Any], rhs: Def[Any]) = rhs match {
    case MatrixNumRows(x) => emitValDef(sym, quote(x) + ".numRows")
    case _ => super.emitNode(sym, rhs)
  }
}
```

Delite Runtime



Schedule & Kernel Compilation

- Compile execution graph to executables for each resource after scheduling
  - Defer all synchronization to this point and optimize
- Kernels specialized based on number of processors allocated for it
  - e.g., specialize height of tree reduction
- Greatly reduces overhead compared to dynamic deferred execution model
  - Can have finer-grained Ops with less overhead

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

Summary

- Delite is a compiler and runtime infrastructure for creating new DSLs embedded in Scala
- Provides a common IR along with generic and parallel optimizations for all DSLs
  - DSL extends IR to add domain ops and domain-specific optimizations
- Domain ops mapped to Delite parallel patterns
  - Delite provides parallel code generation to multiple targets (Scala, C, Cuda) and manages execution

Thank You!

- Questions?

New Features Coming Soon!

Data Structures

- Current: DSL developers implement their own data structures for each target device
- New: everything is a Struct/Record, specified programmatically
  - instantiation and field access lifted into IR

```
def Complex(re: Rep[Double], im: Rep[Double]) =
  new Record { val real = re; val imag = im }
  //lifted to __new and forwarded to Delite

val x = Complex(0,0)
x.real //type-checked that field “real” exists, then forwarded to Delite
```

Why Records?

- We can auto-generate the back-end implementation to different platforms
  - Supported by large number of target languages
- We can reason about which part of the record is actually used
  - Unwrap the record and just pass around required fields in generated code
  - Unused fields can be eliminated all together
- We can perform automatic AoS -> SoA conversion
  - Instantiate only arrays of primitives in the generated code

AoS -> SoA Optimization

- Provide familiar AoS form to the DSL developer
- Perform SoA transformations transparently for DeliteOps
  - Functions returning record types split into result for each component
  - Create separate loop to compute each component
  - Unused components are dead-code-eliminated
  - Loop fusion recombines live components into single loop

Integrating with Control Flow

```
def conj(c: Rep[Complex]) = Complex(c.real, -c.imag)

//a: Rep[Array[Complex]]
val b = if (x > 7) a.map(conj) else a
```

- 'a' exists in IR as ("real" -> Rep[Array[Double]], "imag" -> Rep[Array[Double]])
- a.map is split into map for each component
  - map over real component is optimized away (identity function)
- real array is unaffected by conditional, simply re-used in generated code

Analyses and Transformations

- Current:
  - DSL developers can define IR rewrites using pattern matching as IR is created
  - More complicated analyses & transformations requiring traversals of the full IR require overriding parts of the code scheduler
- New:
  - IR traversal abstraction
  - Pattern matching to find IR nodes of interest
  - Replace node/block with new node/block

Debugging

- Current:
  - Can emit domain-specific error messages during staging
  - But staging approach makes it difficult to report errors that refer back to the DSL application
    - All symbol names are synthetic, no line numbers, etc.
- New:
  - Scala compiler provides SourceContext to the DSL
    - Symbol names, call stack, etc.
  - Delite Cuda generator provides a warning if a Delite Op cannot be generated to Cuda, the reason why, and the source location of the op
  - Profiler tracks execution time, op type, and mapping between source and generated code

Debugging Profiler