Practical reified trees (not only) for GPGPU

@ochafik

http://github.com/ochafik/Scalaxy
http://github.com/ochafik/ScalaCL

Who am I?

- Hobby Scala enthusiast for 4 years
- I hate technology boundaries
  - ScalaCL: runs Scala on graphic cards
  - Scalaxy: macro experiments (faster loops…)
  - JavaCL: Java bindings for OpenCL
  - BridJ: native C / C++ bindings glue
  - JNAerator: native bindings generator

http://ochafik.com

Scaling ScalaCL up

- ScalaCL
  - Runs Scala on GPUs with OpenCL
  - Macro-based: converts Scala AST to C / OpenCL
  - Issue: not modular, not generic
- Reified trees to the rescue
  - Scala AST retained at runtime
  - Assemble and convert to OpenCL at runtime
  - Useful beyond OpenCL

Abstract Syntax Trees (AST)

- What the compiler works with
- Used by DSLs that transform code (expression trees in C# / LINQ)

So you need an AST?

Macros made that easy:

```
import scala.reflect.runtime.universe._
reify { (x: Int, y: Int) => x * y }
```

```
Function(
  List(
    ValDef(Modifiers(PARAM), "x": TermName), IntTpe, EmptyTree),
    ValDef(Modifiers(PARAM), "y": TermName), IntTpe, EmptyTree)),
  Apply(
    Select(Ident("x": TermName)), "$times": TermName)),
    List(Ident("y": TermName)))))
```

Reification is context-aware

```
def buildExpr[A: TypeTag](id: Int) =
  reify { (a: A) => Seq(a, id, typeTag[A]) }
```

Captures free terms + their runtime value

```
buildExpr[Int](10)
```

```
(a: Int) => Seq(a, id /* def value = 10 */, typeTag[Int])
```

Avoid trouble: only capture val / stable paths

Values or their AST, why choose?

```
case class Reified[A](
    value: A,
    expr: Expr[A])

implicit def reified[A](value: A): Reified[A] =
    macro ...

implicit def unwrap[A](reified: Reified[A]): A =
    r.value
```

Capturing reified functions

```
val f = reified { (x: Int) => x * 0.15 }
val g = reified { (x: Int) => x + f(x) }
// With reify, would look like:
// val g = reify { (x: Int) => x + f.splice(x) }

(x: Int) => x + {
  @inline def f(x: Int) = x * 0.15
  f(x)
}
```

Optimizations: val to def, foreach loops

Compiling an AST at runtime

```
import scala.reflect.runtime.universe._
import scala.reflect.runtime.currentMirror
import scala.tools.reflect.ToolBox

val toolbox = currentMirror.mkToolBox()

val expr = reify { (_: Int) * 2 }

val f = toolbox.eval(expr.tree).asInstanceOf[Int => Int]

f(2) == 4
```

Reified values for speed

- Compilation overhead
  - Can start with “normal” values
  - Captures-aware caching

- Runtime specialization + optimizations
  - Akin to C++ templates
  - Beats cold & warm JVM

Building a simple integrator

```
def createIntegrator(step: Double, f: Reified[Double => Double])
  : Reified[(Double, Double) => Double] =
{
  (xMin: Double, xMax: Double) => {
    val nx = ((xMax - xMin) / step).toInt

    var sum = 0.0
    var x = xMin + step / 2
    for (i <- 0 to nx) {
      sum += f(x)
      x += step
    }
    step * sum
  }
}
```

Returns a reified function

Using that integrator

```
val integrator: Reified[(Double, Double) => Double] =
  createIntegrator(
    step,
    // 1 + 2x + 3x^2 + 2x^3
    (x: Double) => 1 + x * (2 + x * (3 + x * 2)))

integrator(0.5, 10.0)                // Direct Scala value
integrator.compile()()(0.5, 10.0) // Recompiled expression
```

- 30% faster once recompiled
- The smaller the functions, the better (microbenchmarks in Scalaxy/Reified, ~ 10x)

Cool, but...

Let’s break from the JVM and see how it helps on GPUs

Back to OpenCL

- OpenGL for general computations
- GPU & CPU implementations
- Portable build / execution toolchain
  - C dialect sources
  - Introspection / binding
  - Scheduling
  - Memory management

ScalaCL

- CLArray[T] stored on GPU
  - primitives
  - tuples / case classes stored fiber by fiber
- Map / filter / reduce operations
  - closures converted to OpenCL
- Best-effort subset: runs if compiles

Familiar “collections”

- Filtering: presence mask + compaction

```
CLFilteredArray[T] =
    CLArray[T] + CLArray[Boolean]
```

- Chained event-based scheduling
  - One write at a time
  - Multiple reads
  - Map / filter return unfinished collections

```
a.map(f).map(g).filter(h)
```

Some impedance mismatch

- OpenCL vs. Scala:
  - Blocks & Tuples
  - Collections runtime
  - Memory allocation
- ScalaCL solutions:
  - Flattening of tuples
  - Collection operations rewritten to while loops

Behind the curtain

```
// Captured and lifted.
int f(int x) {
    return x % 3;
}

kernel void kern(global const int *in, global int *out) {
    size_t i = get_global_id(0);
    out[i] = f(in[i]);
}
```

Matrix multiplication: C = A * B

```
c(i, j) = sum(a(i, k) * b(k, j))
```

```
class Matrix(data: CLArray[Float], rows: Int, cols: Int) {
  def multiply(a: Matrix, b: Matrix): Unit =
    kernel { (i, j) =>
      for (i <- 0 until rows; j <- 0 until cols)
        data(i * cols + j) =
          (0 until a.cols) map (k => {
            a.data(i * a.cols + k) * b.data(k * b.cols + j)
          }).sum
    }
}
```

Leveraging reified: modularity

Used to require inline functions:

```
val in = new CLArray[Int](n)
val out = in.map(x => x % 3)
```

Now we can use functions from elsewhere:

```
val f: CLFunction[Int, Int] = x => x % 3

...
val in = new CLArray[Int](n)
val out = in.map(f)
```

Leveraging reified: Generic

Dynamic typeclass:

- Numeric on steroids
- Erased away by optimizations
- Works in debug mode

```
def divide[N : Generic](a: CLArray[N], b: CLArray[N]) =
  a.zip(b).map(_ / _)

class Matrix[N : Generic](data: CLArray[N], ...)
```

In practice

- Preconvert Scala to OpenCL if possible
  - Spot errors at compilation time
  - Bail out on free types
- Source-based caching of kernels
- Aggressive stream rewrites

```
(0 until n).map(f).filter(g).map(h).sum
```

Try it

```
libraryDependencies +=
"com.nativelibs4java" %% "scalacl" % "0.3-SNAPSHOT"
fork := true // sbt & macros classpath issues
resolvers += Resolver.sonatypeRepo("snapshots")
```

Work in progress, simple examples in tests :-)

Conclusion

- Reified trees improve ScalaCL
  - Better captures
  - Modularity
  - Genericity

    (applicable without OpenCL)
- What's next
  - Reduce, filter, compact from previous versions
  - Capture readonly data structures
  - Support case class in CLArray[T]
- Wanna help?

Questions