ScalaCL

GPGPU-powered collections,
Faster Scala

Olivier Chafik @ ScalaCL

Who am I ?

- Java hobbyist (NativeLibs4Java)
  - Interop. libraries : BridJ, JNAerator
  - GPGPU libraries : JavaCL, ScalaCL
- C++ professional : 3D Graphics, Financial Software… (soon in London)
- Last 6 months in Scala (stillborn startup…)
- Hate Sudoku (too lazy)

What is ScalaCL ?

- A Scala Compiler Plugin
  - Optimizes regular Scala
    (loops on arrays, ranges…)
  - Converts Scala closures to OpenCL code
- GPGPU-powered Parallel Collections
  - Fit in regular collections (.cl ~ .par)
  - Run parallel map, filter… on GPU or CPU
  - Transparently Asynchronous
    (a.map(f).map(g) returns unfinished collection)

Today's topics

- Architecture
  - General loops rewriting
  - OpenCL stuff
    - ScalaCL Collections
    - Scala -> OpenCL Conversion
- Hands-on
  - Reusable parts
  - Building / running
  - Auto tests
  - TODO list

Loops Rewriting : Example

- These equivalent for loops :

```
for (i <- 0 until n) { ... }
(0 until n).foreach(i => { ... })
```

- Can be rewritten to (10x faster) :

```
var ii = 0
while (ii < n) {
  val i = ii
  ...
  ii += 1
}
```

Loops Rewriting : Status

- Limited targets = Array, inline Range
- Many operations, up to x10 faster
  - map, Array.tabulate, foreach, forall…
  - filter, takeWhile…
  - reduce/scan/fold
- A non-intelligent design grown big
  - Widened cases before refactoring
  - Reached limits of initial hack

Loops Rewriting : What's next (1)

- Coalesce maps (+ rewrite as while loops) : views++

```
a.map((_ * 2) + 1)
```

```
a.map(_ * 2).map(_ + 1)
```

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

```
a.map(g.compose(f))
```

- Or anything else :

```
a.
  filter(_ < 10).
  map(_ + 10).
  sum
```

```
var i = 0 ; val n = a.length
var sum = 0
while (i < n) {
  val item = a(i)
  if (item < 10)
    sum += item + 10
}
sum
```

Loops Rewriting : What's next (2)

- New design :
  - Operations streams
    - 1 loop for all (no intermediate collection)
    - Avoid "internal tuples"

```
a.zipWithIndex.map { case (v, i) => v * i }
```

  - Option[T], Seq.apply, Array.apply…
- Foundations are laid down, TODO :
  - Rewrite all operations (STARTED)
  - Detect side-effects (STARTED)
  - Merge functions of non-rewritable collections

Loops Rewriting : New Design

Match sources

- Array[T]
- List[T] (*)
- Inline Range
- *Seq.apply*
- *Option*

Chain *many* operations

- Side-Effects ?
- Traversal order ?
- Expected benefit ?
- Sinks / builders ?

Generate code

- Single while loop (+ output)
- Wire tuple fibers & indexes
- Chain filters & transforms

OpenCL in 1 slide

- Cross-platform execution of C-like code (compiled at run-time)
  - 1-shot or explicit massively parallel
  - Vector types & fast math (implicit parallel / SIMD)
  - CPUs & GPUs with same code
- Allocation of resources (in RAM or VRAM)
  - 1D arrays, 2D & 3D images
  - Share w/ OpenGL
- Asynchronous & chained operations
  - Execution queue, events…

OpenCL's duality : kernel & host

- "kernel" = parallel C function
  - Execution indexes + in/out data
  - Local groups : share memory & concurrency fences…
  - Performance limitations (branches…)
- Host code (C, Java, Python…)
  - Choose implementation (ATI, NVIDIA, Apple, IBM…)
  - Choose devices & create context + queue
  - Allocate resources
  - Enqueue tasks (read, write, executions…) & wait

ScalaCL Collections

- OpenCL-backed collections : CLArray[T], CLRange…
  - Stored in OpenCL buffers
  - Operations = OpenCL kernels
  - Mutable yet asynchronous
    - Next read waits for last writes
    - Next write waits for past reads & writes
  - Pure-Scala debug mode (SCALACL_USE_SCALA_FUNCTIONS=1)
- Use compiler plugin (& supports manual code)
  - Converts Scala closures to OpenCL kernels
  - Blocks unsupported code

ScalaCL Collections : Status

- Tuploid components (no case classes yet) :
  (Int, (Float, Short))
- Simple operations : map, filter, zip, zipWithIndex, size…
- Chained filter + maps :
  - CLFilteredArray[T] (values + presence array)
  - Fast compaction to CLArray[T] (parallel prefix sum)
- map & filter accept complex closures :
  - Tuples, deconstructive assignments, blocks
  - Math & local functions, captured variables
  - Internal loops

Demo

ScalaCL Collections

Feeding the beast : the easy…

- OpenCL requires C code (different from Scala)

```
points.map { case (x, y) => atan(y, x) }
```

Simple conversion of this closure :

```
kernel void angleMap(
    global const double* x, // first fiber
    global const double* y, // second fiber
    global double* out
) {
    int i = get_global_id(0);
    out[i] = atan2(y[i], x[i]);
}
```

Feeding the beast : the less easy

- Valued blocks, tuples & local functions = alien to C

```
v => {
  val pair @ (a, b) =
    if (v < 10) {
      def sq(x: Int) = x * x
      (v, { val vv = sq(v) ; vv - v })
    } else
      (0, if (v > 100) v - 100 else v)
  (pair, Seq(a, b, v).sum)
}
```

OpenCL Conversion : Normal Form

- Limitations on Scala syntax to make it convertible
  - No tuples
  - No Options
  - Functions : scala.math & locally *accessible*
  - Collections : only constant Array & ranges
  - No classes (TODO lift this one !)
- Most code can be reduced to
  - Outer Declarations (functions, classes…)
  - Local Declarations
  - Return Values (flattened tuple fibers)

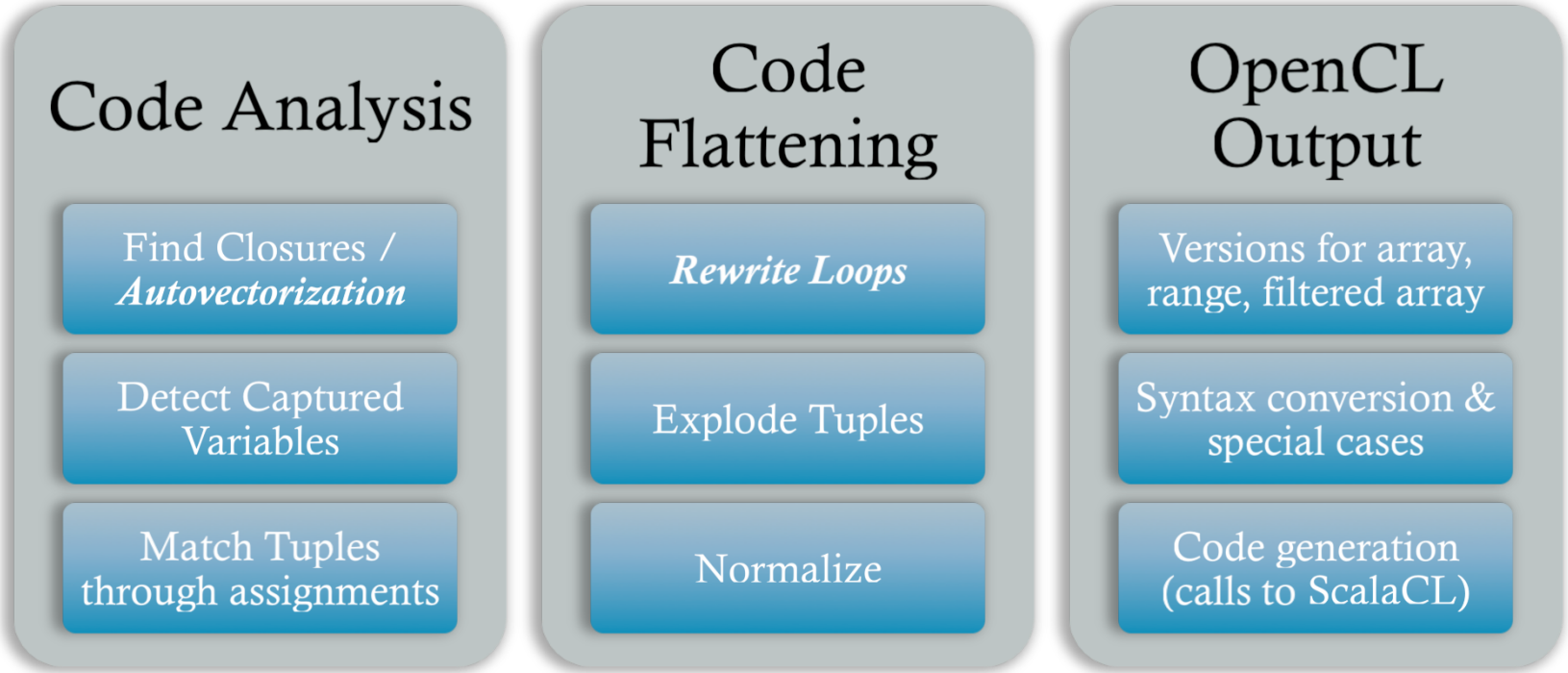
Code Flattening : Example

```
val (a, (b, c)) = {
  def test(x: Int) = x < 10
  if (test(v)) {
    val d = v * 10
    (10, (20, d - 1))
  } else
    (20, (100, 0))
}
```

To see OpenCL result :
SCALACL_VERBOSE=1

```
outerDeclarations = Seq(
  def test(x: Int) = x < 10
)
declarations = Seq(
  var a = 0,
  var b = 0,
  var c = 0,
  var cond = test(v),
  var d = 0,
  if (cond) { d = v * 10 }
)
values = Seq(
  if (cond) 10 else 20, // cond ?
  if (cond) 20 else 100,
  if (cond) d - 1 else 0
)
```

OpenCL Conversion : Overview



Demo

Generated OpenCL Code

Reusable Compiler Plugin Parts

- Lots of matchers (ScalaCLPlugin/…/MiscMatchers.scala)
  - inline Range, ArrayOps, Array, List, Seq…
  - Tuples
- Generators that care about symbols more than TreeDSL (ScalaCLPlugin/…/TreeBuilders.scala)
  - While loops (obviously)
  - Variable definitions
  - Symbol replacements
- Code Analysis (tuple info, side-effects…)

Building / running ScalaCL

- Two Maven2 projects : ScalaCL & ScalaCLPlugin
  - Depends on JavaCL -> BridJ
  - sbt : some quirks…
- scalacl.plugin.Compile
  mvn compile scala:run -DmainClass=scalacl.plugin.Compile "-DaddArgs=Test.scala"
- JUnit tests
  SCALACL_TEST_PERF=1 mvn test
- sbaz deployment :
  ScalaCL/sbazPackage

ScalaCL Tests

Compile & run code snippets with plugin !

- Bytecode Tests
  - Compare to manual rewrite
  - Check unchanged cases
  - Misc bugs tests
- Performance Tests ("*> x times faster*")
  - Enable with SCALACL_TEST_PERF=1
  - Abandoned "optimizations" : SCALACL_EXPERIMENTAL=1
- Runtime Tests
  - Compare results of map, filter… on Array and CLArray
  - Capture of scalars and arrays
  - Test of run-time code generation

ScalaCL Tests : TODO

- Fix Scala 2.9.0 migration
  - Cannot reuse compiler anymore (tests 4 x slower !)
- Fix sbt integration + migrate to 0.10.0
- Move to ScalaTest
- Use ScalaCheck ? (chaining operations…)
- More tests & their fixes ;-)
  - Side-effects detection
  - Weird tuples syntaxes vs. OpenCL conversion

ScalaCL : TODO

- General Loops Rewriting
  - 1 bug (manifests…)
  - Finish implementation of new design
  - Side-effects detection to allow chaining (STARTED)
- ScalaCL Collections
  - Copy-on-write cloning (STARTED)
  - reduceSymmetric (OpenCL) + symmetry detection (Compiler Plugin)
  - “pure” case classes components
- OpenCL conversion
  - 1 bug (tuple alias)
  - Homogeneous tuples + DSL
  - Constant Array
  - Synthetize cached function objects
  - Auto-vectorization (CLArray.apply seeds -> ranges)
  - Generate all code during compilation (STARTED)