Delite

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

Tiark Rompf, Martin Odersky
EPFL
Administrative

- PS 1 due today
  - Email to me

- PS 2 out soon
  - Build a simple DSL in LMS + Delite
Where we left off...

/**************************************************************************
Emitting Generated Code
**************************************************************************/

class Application extends ((Unit)=>(Unit)) {
    def apply(x0:Unit): Unit = {
        val x1 = new MatrixImpl(10,10)
        val x2 = x1 * x1
        val x3 = x2 * x1
        val x4 = x3 * x1
        val x5 = x4 * 1
        val x6 = println(x5)
        x6
    }
}

/**************************************************************************
End of Generated Code
**************************************************************************/
What about parallelism?

- **Solution 1:**
  - LMS gives you the power to generate arbitrary code: make it parallel
    - Upside: can do anything you want
    - Downside: have to do it

- **Solution 2:**
  - Add a middle layer between LMS and the DSL that deals specifically with parallelism
    - This is Delite
Parallelism within an Operation

- **Before:**
  - case class DSLop extends Def

- **After:**
  - case class DSLop extends DeliteOp
  - trait DeliteOp extends Def
  - DSLop expresses domain information
  - DeliteOp expresses parallelism information
**Delite Ops**

- Express known parallel patterns
  - Right now data-parallel ones
  - Map, Zip, Reduce, MapReduce, ZipReduce

- MultiLoop
  - Parallel loop with disjoint accesses
  - + optional reduction
  - Enables loop fusing optimization
    - Annoyance: have to add a mirror function for every IR node

- Foreach
  - Allows non-disjoint accesses

- SingleTask
  - Arbitrary, sequential code
trait DeliteOp[A] extends Def[A]

trait DeliteOpMap[A,B,C[X] <: DeliteCollection[X]] extends DeliteOp[C[B]] {
  val in: Exp[C[A]]
  val v: Sym[A]
  val func: Exp[B]
  val alloc: Exp[C[B]]
}

  val alloc = reifyEffects(Vector[A](in.length, in.isRow))
  val v = fresh[A]
  val func = reifyEffects(v + y)
}
abstract class DeliteOpForeach[A,C[X] <: DeliteCollection[X]]
  extends DeliteOp[Unit] {
    val in: Exp[C[A]]
    val v: Sym[A]
    val func: Exp[Unit]
    val i: Sym[Int]
    val sync: Exp[List[Any]]
  }

case class VerticesForeach[V:Manifest](in: Exp[Vertices[V]], v: Sym[V],
  func: Exp[Unit]) extends DeliteOpForeach[V,Vertices] {
    val i = fresh[Int]
    val sync = reifyEffects(in(i).neighborsSelf.toList)
  }

def vertices_foreach[V:Manifest](x: Exp[Vertices[V]], block: Exp[V] => Exp[Unit]) = {
  val v = fresh[V]
  val func = reifyEffects(block(v))
  reflectEffect(VerticesForeach(x, v, func))
}
abstract class AbstractLoop[A] extends Def[A] {
  val size: Exp[Int]
  val v: Sym[Int]
  val body: Def[A]
}


case class DeliteCollectElem[A, CA <: DeliteCollection[A]](
  alloc: Exp[CA],
  func: Exp[A]
) extends Def[CA]

case class DeliteReduceElem[A](
  func: Exp[A],
  cond: List[Exp[Boolean]] = Nil,
  zero: Exp[A],
  rV: (Sym[A], Sym[A]),
  rFunc: Exp[A]
) extends Def[A]
abstract class AbstractLoop[A] extends Def[A] {
    val size: Exp[Int]
    val v: Sym[Int]
    val body: Def[A]
}

case class DeliteCollectElem[A, CA <: DeliteCollection[A]](
    alloc: Exp[CA],
    func: Exp[A]
    cond: List[Exp[Boolean]] = Nil
) extends Def[CA]

    extends DeliteOpLoop[Vector[A]] {
    val size = inA.length
    val v = fresh[Int]
    val body = DeliteCollectElem[A, Vector[A]](
        alloc = reifyEffects(Vector[A](size)),
        func = reifyEffects(inA(v) + inB(v))
    )
```scala
case class DeliteReduceElem[A](
    func: Exp[A],
    cond: List[Exp[Boolean]] = Nil,
    zero: Exp[A],
    rV: (Sym[A], Sym[A]),
    rFunc: Exp[A]
) extends Def[A]

val body = DeliteReduceElem[A](
    func = reifyEffects(in(v)),
    zero = getMaxValue[A],
    rV = rV,
    rFunc = reifyEffects(if (rV._1 < rV._2) rV._1 else rV._2)
)
```
abstract class AbstractLoop[A] extends Def[A] {
  val size: Exp[Int]
  val v: Sym[Int]
  val body: Def[A]
}

case class DeliteCollectElem[A, CA <: DeliteCollection[A]](
  alloc: Exp[CA],
  func: Exp[A]
  cond: List[Exp[Boolean]] = Nil
) extends Def[CA]

class VectorFilter[A:Manifest](in: Exp[Vector[A]],
  val size = in.length
  val v = fresh[Int]
  val body = new DeliteCollectElem[A,Vector[A]](
    alloc = reifyEffects(Vector(0,in.isRow)),
    func = reifyEffects(in(v)),
    cond = reifyEffects(pred(in(v))::Nil
  )
}
trait DeliteOpMap[A, B, C[X]] <: DeliteCollection[X] extends DeliteOp[C[B]] {
  val in: Exp[C[A]]
  val v: Sym[A]
  val func: Exp[B]
  val alloc: Exp[C[B]]
}

trait DeliteCollection[T] {
  def size: Int
  def dcApply(idx: Int): T // apply with a flat, 1D view of the collection
  def dcUpdate(idx: Int, x: T)
}

class Matrix[T:Manifest](numRows: Int, numCols: Int) extends DeliteCollection[T] {
  val _data = new Array[T](size)
  def size = numRows*numCols
  def apply(i: Int, j: Int): T = _data(i*numCols+j)
  def update(row: Int, col: Int, x: T) { _data(row*numCols+col) = x }
  def dcApply(idx: Int): T = _data(idx)
  def dcUpdate(idx: Int, x: T) { _data(idx) = x }
}
DeliteOp Code Generation

- The DeliteOp type provides a clear parallelization strategy for each target
  - Delite handles the codegen for all targets (Scala, Cuda) for all DeliteOps
    - Should succeed as long as the function can be generated for the target
override def emitNode(sym: Sym[Any], rhs: Def[Any])(implicit stream: PrintWriter) = rhs match {
  case map:DeliteOpMap[_,_,_] => {
    if(!isPrimitiveType(map.func.Type)) throw new GenerationFailedException("CudaGen: Only primitive Types are allowed for map.")
    if(!isPrimitiveType(map.v.Type)) throw new GenerationFailedException("CudaGen: Only primitive Types are allowed for map.")
    currDim += 1
    val currDimStr = getCurrDimStr()
    setCurrDimLength(quote(map.in)+"->size()")
    stream.println(addTab() + "if( %s < %s ) {".format(currDimStr,quote(map.in)+".size()"))
    tabWidth += 1
    val (mapFunc,freeVars) = emitDevFunc(map.func, List(map.v))
    if(freeVars.length==0)
      stream.println(addTab() + "%s.dcUpdate(%s, %s(%s.dcApply(%s)));".format(quote(sym),currDimStr,mapFunc,quote(map.in),currDimStr))
    else
      stream.println(addTab() + "%s.dcUpdate(%s, %s(%s.dcApply(%s),%s));".format(quote(sym),currDimStr,mapFunc,quote(map.in),currDimStr,freeVars.map(quote).mkString(",",))
      tabWidth -= 1
      stream.println(addTab() + "}"
    emitAllocFunc(sym,map.alloc)
    if(map.in==map.alloc) throw new GenerationFailedException("CudaGen: Mutable input is not supported yet.")
    currDim -= 1
  }
}
Aside: Performance Considerations

- HotSpot uses a method-based JIT compiler
  - Inlining everything into one big method is a recipe for very slow code

- We have no notion of any user-defined or DSL-defined methods
  - So we emit every DeliteOp as a method
  - If you add any non-trivial code generation you should keep this in mind
DeliteApplication

- Provides the “main” method for the DSL compiler
  - Initializes all target code generators
  - Creates the “generated” directory for all the generated kernels and data structures
  - Constructs the IR and passes it to DeliteCodeGen
- Should be extended by the DSL application runner

```scala
trait DeliteApplication {
  final def main(args: Array[String]) { ... }

  var args: Rep[Array[String]] = _ //args accessed through field

  def main() //the DSL application’s main method, calling it builds the IR
}
```
Parallelism Among Operations

- LMS tracks all dependencies among IR nodes to schedule code

- Delite uses the information to discover task parallelism
  - Export all dependency information for each Op to create the Delite Execution Graph (DEG) file
  - The DEG contains all the information the runtime needs to execute the application
Delite Code Generator

- Just another code generator with emitNode()
  - Writes the Op information into the DEG
  - Creates a kernel file header and then calls emitNode() for every registered target generator (Scala, Cuda)
    - Every Op at the top-level of the program is a kernel, nested calls to emitNode yield inlined code within kernel
  - If a target generator throws a GenerationFailedException, doesn’t include that target in the list of choices for this Op

- Delite has a single view of the IR schedule and multiple code generators for each node
  - Therefore each generator must agree completely on the code schedule
"type":"MultiLoop", "needsCombine":true, "kernelId":"x491x506x521",
"supportedTargets": ["scala"],
"outputs": ["x491","x506","x521"],
"inputs": ["x257","x477","x128","x398"],
"mutableInputs": [],
"controlDeps": [],
"antiDeps": [],
"metadata": {},
"return-types": {"scala" : "activation_x491x506x521"},
"output-types": {"x491": {"scala": "Double"}, "x506": {"scala": "Double"},
"x521": {"scala": "Double"}}
}
Op with Scala & Cuda

```json
{
    "type": "Map",
    "kernelId": "x83",
    "supportedTargets": ["scala", "cuda"],
    "outputs": ["x83"],
    "inputs": ["x75", "x79", "x80"],
    "mutableInputs": [],
    "controlDeps": ["x6", "x52", "x75", "x85"],
    "antiDeps": [],
    "metadata": {
        "cuda": {
            "gpuBlockSizeX": ["gpuBlockSizeX_x83_4", ["x83", "x75", "x79", "x80"]],
            "gpuBlockSizeY": ["gpuBlockSizeY_x83_4", ["x83", "x75", "x79", "x80"]],
            "gpuBlockSizeZ": ["gpuBlockSizeZ_x83_4", ["x83", "x75", "x79", "x80"]],
            "gpuDimSizeX": ["gpuDimSizeX_x83_4", ["x83", "x75", "x79", "x80"]],
            "gpuDimSizeY": ["gpuDimSizeY_x83_4", ["x83", "x75", "x79", "x80"]],
            "gpu Inputs": {
                "x75": ["Vector<double>", "copyInputHtoD_x83_x75_2",
                          "copyMutableInputDtoH_x83_x75_3"]],
                "gpuOutput": {
                    "x83": ["Vector<bool>", "allocFunc_1",
                             [x79, x80], "copyOutputDtoH_1", ["env", "x83"]],
                        "gpuTemps": []}
                "output-types": {
                    "x83": {
                        "scala": "generated.scala.Vector[Boolean]",
                        "cuda": "Vector<bool>"}
                }
            }
        }
    }

```
Nested Graphs

A nested graph consists of:

- A conditional graph with:
  - Output ID: "x6"
  - Condition type: "symbol"
  - Condition operations:
    - "EOG"
  - Then type: "symbol"
  - Then operations:
    - "EOG"
  - Else type: "const"
  - Else value: "()"

- Control dependencies: []
- Anti dependencies: []

- Return types:
  - Scala: "Unit"
  - C: "void"
Map Kernel

```scala
package generated.scala
final class activation_x83 { // generated even if not used
    var x83: generated.scala.Vector[Boolean] = _
}

object kernel_x83 {
        val x83 = new generated.scala.DeliteOpMap[Double, Boolean, generated.scala.Vector[Boolean]] {
            def in = x75
            def alloc = {
                val x81 = new generated.scala.BooleanVectorImpl(x79, x80)
                x81
            }
            def map(x76: Double) = {
                val x77 = x76 <= 0.0
                val x78 = {
                    def x78thenb(): Boolean = false
                    def x78elseb(): Boolean = true
                } if (x77) { x78thenb() } else { x78elseb() }
            }
        }
        x83
    }
}
```
Delite Overrides

- Delite overrides the LMS implementation of a few Scala constructs
  - You better mix in DeliteAllOverridesExp

- Example: Variables
  - If a variable escapes a single kernel there’s no way to update the reference
    - Method arguments are immutable
  - Solution: Convert the variable to an object with a mutable field
Delite Runtime

- A common runtime for Delite-based DSLs

- Maps the DEG onto the current machine
  - Really onto the configuration you tell it
    - -Ddelite.threads=n to use n cores
    - -Ddelite.gpus=1 to use a Cuda GPU

- Responsible for all execution details
  - synchronization, data transfers, memory management, etc.

- Responsible for compiling all the generated code
  - Yet another possible stage to find compiler bugs
  - Caches compiled code for faster subsequent runs
Static Scheduling

- Use execution graph to create a static schedule for the machine

- Control flow handled with nested graphs
  - e.g., *While* is a node in the outer graph, and contains a graph for evaluating it’s predicate and another for it’s body
  - Schedule each graph independently
object Executable0 extends DeliteExecutable { //launched on thread 0
    def run() {
        val x1 = kernel_1()
        val x2 = Executable1.get_x2
        val x3 = kernel_3(x1,x2)
    }
}

object Executable1 extends DeliteExecutable { //launched on thread 1
    def run() {
        val x2 = kernel_2()
        Result2.set(x2)
    }

    def get_x2 = Result2.get

    object Result2 { 
        var result = null 
        def get ... //block until result available, read is destructive
        def set ... //store result, block until result is empty
    }
}
Kernel Compilation

- Data-parallel ops can be split across multiple processors (as chosen by the scheduler)

- Part of the kernel is generated by the runtime after scheduling

- e.g., Reduce Op
  - Compiler generates the reduction function
  - Runtime generates a kernel for each processor that performs a tree-reduce with levels determined by the processor count
Delite / Cuda

- Runtime generates a Cuda host thread to launch kernels and perform data transfers
  - Inputs not already on GPU (or not valid) transferred there right before kernel launch
    - Uses the schedule to determine if an input has been mutated by the CPU between when it was initially transferred and the current time
  - Output (and any mutated inputs) transferred back to CPU upon completion if required
  - Synchronization with CPU threads done on the Scala side (through JNI)
  - Compiler provides helper functions that the runtime calls to copy data structures, pre-allocate outputs and temporaries, and select the number of threads & thread blocks
GPU Memory Management

- Runtime provides a malloc() function which is used by the compiler-generated helper functions
  - Registers all memory allocations and associates each with the Op that caused it

- Uses the DEG and schedule to perform liveness analysis and determine when each piece of data will no longer be needed on the GPU
  - When a piece of data becomes dead it is added to a free list

- The Cuda host thread uses asynchronous memory transfers & kernel launches to run ahead of the GPU as much as possible; when the GPU runs out of memory it blocks
  - Waits for the Op associated with the next item on the free list to complete and performs a free
Check out the new webpage!