DFL
A Data Flow Language to Develop High Performance Computing DSLs

Alejandro Fernández, Vicenç Beltran* and Eduard Ayguadé
{afernand, vbeltran, eduard.ayguade}@bsc.es

New Orleans, Louisiana, USA
November 17th, 2014
**Motivation**

- **Domain-Specific Languages (DSLs)**
  - Hide the complexity of HPC systems
  - Boost programmer’s productivity

- **DSL drawbacks**
  - High development cost due to implementation complexity
  - Efficiency and high scalability are a must

- **Our proposal**
  - Provide a common DSL development infrastructure
  - Amortize its cost by implementing many HPC DSLs with it
Underlying Technologies

**HPC Execution Framework**
- **OmpSs programming model**
  - High-level, task-based, parallel programming model supporting SMPs, heterogeneous systems and clusters
  - Coupled with its Nanos++ runtime system, is ideal as target language for the DSL framework

**Compilation Framework**
- **Scala**
  - Statically typed, multi-paradigm language with type inference
  - Supports both functional and object-oriented styles
- **Lightweight Modular Staging (LMS)**
  - A technique for embedding DSLs as Scala libraries
  - Enables domain-specific optimizations and code generation
**Data-Flow Language (DFL)**

- A DSL meant to be used as target language for HPC DSLs implemented with Scala and LMS
- Wraps up OmpSs features

**Main Features**

- Tasks and Kernels
- Buffers & Distributed buffers
- High-level parallel, distributed operations
  - Map
  - Reduce
  - Divide & Conquer
**Buffers**
- Generic data containers

```
// Allocate space for 4096 floating point values
val buf1 = Buffer.fill[Float](4096)
// Buffer of 4 integer values
val buf2 = Buffer(5, 8, 21, -3)
```

**Distributed Buffers**
- Generic data containers for **distributed** environments
- Provide high level operations

```
// 4096 Floats distributed among all nodes
val world = MPI_COMM()
val dbuf = DistBuffer.fill[Float](world, 4096)
dbuf map { f => sqrt(f) }
dbuf.rotateLeft
val result = dbuf allReduce +
```
Buffers

- Generic data containers
  
  // Allocate space for 4096 floating point values
  val buf1 = Buffer.fill[Float](4096)
  
  // Buffer of 4 integer values
  val buf2 = Buffer(5, 8, 21, -3)

Distributed Buffers

- Generic data containers for **distributed** environments
  
  // 4096 Floats distributed among all nodes
  val world = MPI_COMM()
  val dbuf = DistBuffer.fill[Float](world, 4096)
DFL – Tasks and Kernels

Tasks
- Computational function with annotated parameters
- Parameters can be of any type

```
Task(A, B)(In, InOut) {
    B += A;
}
```

Kernels
- Tasks (written in OpenCL C) for accelerators (GPUs, Intel’s Xeon Phi, etc)
- Parameters can be primitive types or Buffers

```
val kc = KernelContainer("/path/to/kernels.cl")
// This retrieves the 3-parameter "add" kernel
// from kc, namely myAdd
val myAddKernel = Kernel(kc, "add")(In, In, Out)
myAddKernel(A, B, C)
```
**DFL – High Level Operations**

**Map**
- Applied locally or at a distributed level, depending on buffer type
  
  ```scala
  val b = Buffer.fill(4096*4096)
  B map { _ => rand }
  
  val db = DistBuffer.fill(world, 4096*4096) // Collective operation
  db map { x => sqrt(x) } // Collective operation
  ```

**Reduce**
- Fold a buffer with a binary operator and accumulate the result
  
  ```scala
  val result = db.allReduce{ (x,y) => max(x/2, y+5) }
  ```

**Divide & Conquer pattern**
- Split a problem into smaller subproblems, solve them and combine the solutions in a potentially distributed environment (see next slides)
DFL – Divide and Conquer

Divide

- A function that partitions the problem if its size is bigger than a certain threshold (base case size)

```scala
val divFun = { p =>
  if (p.size > BASE_CASE_SIZE) {
    val chunkSize = p.size / 2
    // setRange is a shallow copy with just range override
    val pleft = p.setRange(p.begin, p.begin+chunkSize)
    val pright = p.setRange(p.begin+chunkSize, p.end)
    List(pleft, pright)
  }
  else List(p)
}
```
**Solve**

- A function that solves a problem given its size

```scala
val solveFun = { p =>
  sort(p.data, p.begin, p.end)
}
```
Combine

- A function that combines a list of solved problems into a whole solution

```scala
val combineFun = { xl =>
    xl reduceLeft { (sorted, ls) => merge(sorted, ls) }
}
```
## Execution

- Initialize a distributed problem, solve it with divide a conquer on each node, combine the distributed solution

```scala
val world = MPI_COMM()
val data = new DistBuffer[Int](world, 4096)
val tmp = new DistBuffer[Int](world, 4096)
data map { _ => rand }
data divConquer(divFun, solveFun, combineFun)
(0 until world.size) foreach { _ =>
    // Pairwise exchange and function application
    data exchangeAndApply(tmp, merge)
}
show(data)
```
Conclusions & Future Work

DFL is a DSL designed to exploit distributed and heterogeneous HPC systems
- Serves as the target language for other DSLs, enabling simple code generators without sacrificing HPC performance
- Leverages the LMS framework for the DSL compiler infrastructure and the hybrid MPI/OmpSs programming model the DSL runtime systems

Interoperability
- DFL can enable DSL interoperation via a convenient infrastructure, which will also enable reuse of different DSL implementations, not just the DFL infrastructure
Thank you!

For further information please contact
pm@bsc.es
CASE expertise on Partial Differential Equations and HPC
  – Alya Red

Domain: Convection-Diffusion-Reaction equations
  – Well know domain (by the CASE people)
  – Several implementations already available in C and Fortran
  – First design decisions of the DSL
    • Level of abstraction
    • Types
    • Operators

CS / CASE collaboration

WOLFHPC, New Orleans, LA, November 17th, 2014
Saiph: A domain specific language for solving PDEs

- Simple and high level syntax
  - High level constructs that directly associate with domain knowledge
  - Efficient development/maintenance cycle
- High performance computing for free (for the end user)
  - Ability to solve large complex problems with 20 lines of clean, simple code

This is a program that runs on a **GPU**: 10.000 time steps in 7 seconds

```-scala
val c = Cartesian(12.5, 25.0, 37.5)
val temp = Unknown(c)
val cond = Dirichlet(lowXZ of c, temp, 400)
val hv = Vector(0.5, 0.5, 0.5)
val pre = PreProcess(nsteps = 10000, deltaT = 0.25, h = hv)(cond)
solve(pre) equation (0.15 * lapla(temp) - dt(temp)) to "diffusion"
```
def KFun(xp: Float, yp: Float, zp: Float) = {
    if (zp > 18.75) 0.02
    else 0.15
}

val c = Cartesian(12.5, 25.0, 37.5)
val temp = Unknown(c)
val plane = Dirichlet(lowXZ of c, temp, 400)

val hv = Vector(0.5, 0.5, 0.5)
val pre = PreProcess(nsteps = 100000, deltaT = 0.125, h = hv)(plane)

val K = KFun _
val diffusion = K * lapla(temp) - dt(temp)

val post = snapshot each 100 steps

solve(pre)(post) equation diffusion to "diffusion"
CDR: Example 1 – Pure diffusion phenomena
CDR: Example 1 – Pure diffusion phenomena

Saiph generates
- Two OpenCL kernels (tasks)
- One I/O task
- The initialization code + body of the application + OmpSs pragmas

OmpSs runtime orchestrates the execution
- Schedules task based on data dependencies
- Manages data transfers between host and GPU
def hotCube(cx: Float, cy: Float, cz: Float, edgeSize: Float)  
(xp: Float, yp: Float, zp: Float) = {
  if (xp >= cx - edgeSize && xp <= cx + edgeSize &&
      yp >= cy - edgeSize && yp <= cy + edgeSize &&
      zp >= cz - edgeSize && zp <= cz + edgeSize)  Some(10)
  else Some(5)
}

val c = Cartesian(25, 50, 75)
val temp = Unknown(c)
val cube = Source(hotCube(12.5, 25, 37.5, 6) _, temp)

val hv = Vector(1, 1, 1)
val pre = PreProcess(nsteps = 500, deltaT = 1, h = hv)(cube)(PeriodicHighZ)

val v = Vector(0, 0, 1)
val convection = dt(temp) + grad(temp) * v
solve(pre)(flush) equation convection to "convection"
The numerical scheme do not introduce artificial diffusion due to the stabilization. The cubic form is preserved.
def CDef(x: Rep[Float], y: Rep[Float], z: Rep[Float]) = {
    if (x >= 300 && x <= 400 && y >= 300 && y <= 400) (1700*1700)
    else (2000*2000)
}
val c = Cartesian(500, 500, 9)
val pressure = Unknown(c)
val waveSource = PointSourceSource(250, 250, 5)(rickerWalet(20), pressure)

val hv = Vector(1, 1, 1)
val pre = PreProcess(nsteps = 50000, deltaT = 0.003333, h = hv)(waveSource)
val C = CDef _

val wavePropagation = C * lapla(pressure) – dt2(pressure)

val post = snapshoot each 10 steps
solve(pre)(post) equation wavePropagation to "wave"
Example 4 – Acoustic wave equation
Complete code in backup slides

val c = Cartesian(125, 250, 375)
val temp = Unknown(c)
val tori = Source(MyTori _, temp)

val hv = Vector(0.95, 0.95, 0.95)
val pre = PreProcess(nsteps = 10000, deltaT = 0.5, h = hv)()(tori)

val K = KVarFun _
val v = Vector(0.05, 0.05, 0)

val heat = K * lapla(temp) + grad(temp) * v - dt(temp)

val post = snapshot each 200 steps

solve(pre)(post) equation heat to "toriHeat"
Example 3 – Heat convection and diffusion using toroidal sources
Underlying Technologies

CDRs Embedded Compiler (LMS)

Diffusion.rsv ➔ Scala Virtualized Compiler ➔ Diffusion.class

Host-side CodeGen ➔ Accelerator-side CodeGen

CDRs Embedded Compiler (LMS)

Diffusion.dfl ➔ DiffusionEquation.rsveq ➔ Equation Stencil Compiler (LMS)

DiffusionKernels.cl ➔ DFL Compiler (LMS)

Diffusion.cpp ➔ OmpSs

Front end
- Compile the program with the LMS Library and the compiler implementation together

Middle end
- 1st stage
- Domain Specific Opt.
- LMS IR generation

Back end
- 2nd stage
- DFL code + OpenCL kernels