Implementing domain-specific languages for heterogeneous parallel computing

Luca Baroffio
Parallel programming

• Needed due to the advent of multi-core processors and to exploit specialized hardware components.

Three main characteristics aka 3 Ps:

• Productivity
• Performance
• Portability
Domain Specific Languages

• Focus on performance and productivity trading off generality;
• Ideal platform to tackle the parallel programming problem

Internal DSLs ("just-a-library")
• Easy to implement;
• Difficult to optimize
• No code generation for heterogeneous devices.

External DSLs ("stand-alone")
• Difficult to implement;
• Powerful and flexible;
• Need an entire compiler framework and a set of tools.

Language virtualization
Hybrid approach; exploits a flexible host language to provide embedded DSLs indistinguishable from stand-alone ones.
Delite compiler and runtime framework

• A set of common **parallel execution patterns** to ease the development of DSLs;
• **Code generators** to target heterogeneous hardware (multi-core CPU, GPU, ..);
• **Runtime management**, including scheduling and synchronization.
Delite framework operations

```scala
// x : TrainingSet[Double]
// mu0, mu1 : Vector[Double]
val sigma = sum(0, m) { i =>
    if(x.labels(i) == false){
        ((x(i)-mu0).t) ** (x(i)-mu0)
    } else{
        ((x(i)-mu1).t) ** (x(i)-mu1)
    }
}
```

OptiML application

Delite compiler framework

Delite runtime
• **OptiML**, a machine learning DSL;
• Provides domain specific methods, data structures and operations;
• Developed starting from the **patterns** provided by the Delite framework

```scala
// x : TrainingSet[Double]
// mu0, mu1 : Vector[Double]

val sigma = sum(0, m) { i =>
  if(x.labels(i) == false) {
    ((x(i)-mu0).t) ** (x(i)-mu0)
  }
  else {
    ((x(i)-mu1).t) ** (x(i)-mu1)
  }
}
```
Delite compiler

• Generate an **Intermediate Representation**;
• Code analysis and optimization performed on the IR;
• **Delite Execution Graph**, contains information about how to schedule the kernels;
• Generate **computation kernels** for the hardware (CPUs, GPU, ...)

```
Build IR

Initial IR

Common subexpression elimination (CSE)
  x(i)-mu0
  x(i)-mu1

Linear algebra simplification

Further IR optimizations

Optimized IR

DEG generator

Scala generator

C++ generator

CUDA generator
```
Delite runtime

- Collects machine specifications and other input data;

- Generates an execution plan for each available resource starting from the DEG;

- Target language compilers build the code according to the execution plan to generate executable files.
Building embedded parallel DSLs
Intermediate Representation (IR)

• The IR is **optimized** at different layers (views):
  
  – **Domain Specific IR view**: apply domain specific optimization (e.g. linear algebra optimization);
  
  – **Parallel IR view**: provides parallel execution templates and optimization (e.g. DeliteOpMap, DeliteOpReduce, loop fusing and peeling);
  
  – **Generic IR view**: common static optimization (e.g. Common Subexpression elimination).

• DSL developers only need to **extend parallel IR nodes provided by the framework** to implement methods and data structures.
For each IR node the Delite compiler framework generates kernels in multiple target programming languages (Scala, C++, CUDA);

The decision on which kernel to be executed is deferred until runtime;

GPU code generation is a critical task: no dynamic memory allocation, the CUDA generator preallocates static memory before the execution, based on memory requirements metadata stored in the DEG.
Executing embedded parallel DSLs

• The Delite Runtime combines the DEG with machine specifications to **schedule the application across the available resources**;

• Each resource has its own **execution plan** made by executable files. Besides the computation tasks, each kernel is also responsible for the necessary **synchronization**.

• The runtime framework also take care of **memory allocation** in different address spaces.
Performance evaluation

---

(a) Normalized execution time:

- 1 CPU: 1.0
- 2 CPUs: 1.6
- 4 CPUs: 1.8
- 8 CPUs: 1.9
- CPU + GPU: 2.6

(b) Normalized execution time:

- 1 CPU: 1.0
- 2 CPUs: 1.2
- 4 CPUs: 1.3
- 8 CPUs: 1.8
- CPU + GPU: 8.1

(c) Normalized execution time:

- 1 CPU: 1.0
- 2 CPUs: 2.1
- 4 CPUs: 4.1
- 8 CPUs: 7.1
- CPU + GPU: 2.3

(d) Normalized execution time:

- 1 CPU: 1.0
- 2 CPUs: 1.9
- 4 CPUs: 3.1
- 8 CPUs: 4.2
- CPU + GPU: 1.1

Legend:
- OptiML
- Parallelized Matlab
- C++
• Implementing domain-specific languages for heterogeneous parallel computing

HyoukJoong Lee, Kevin J. Brown, Arvind K. Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, Kunle Olukotun

*IEEE Micro: Special Issue on CPU, GPU, and Hybrid Computing, September/October 2011.*