A Heterogeneous Parallel Framework for Domain-Specific Languages

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

Tiark Rompf, Martin Odersky
EPFL
Programmability Chasm

Applications

- Scientific Engineering
- Virtual Worlds
- Personal Robotics
- Data Informatics

Parallel Programming Language

- Pthreads
- OpenMP
- CUDA
- OpenCL
- Verilog
- VHDL
- MPI
- PGAS

Platforms

- Sun T2
- Nvidia Fermi
- Altera FPGA
- Cray Jaguar
The Ideal Parallel Programming Language
Successful Languages

- Performance
- Productivity
- Generality

Languages:
- C/C++
- Java
- Python
- Ruby
Domain Specific Languages

Performance (Heterogeneous Parallelism)

Productivity

Generality

Domain Specific Languages

SQL

MATLAB

C/C++

Python

Ruby
Benefits of Using DSLs for Parallelism

Productivity
- Shield most programmers from the difficulty of parallel programming
- Focus on developing algorithms and applications and not on low level implementation details

Performance
- Match high level domain abstraction to generic parallel execution patterns
- Restrict expressiveness to more easily and fully extract available parallelism
- Use domain knowledge for static/dynamic optimizations

Portability and forward scalability
- DSL & Runtime can be evolved to take advantage of latest hardware features
- Applications remain unchanged
- Allows innovative HW without worrying about application portability
DSLs: Compiler vs. Library

- A Domain-Specific Approach to Heterogeneous Parallelism, Chafi et al.
  - A framework for parallel DSL libraries
  - Used data-parallel patterns and deferred execution (transparent futures) to execute tasks in parallel

- Why write a compiler?
  - Static optimizations (both generic and domain-specific)
  - All DSL abstractions can be removed from the generated code
  - Generate code for hardware not supported by the host language
  - Full-program analysis
Common DSL Framework

- Building a new DSL
  - Design the language (syntax, operations, abstractions, etc.)
  - Implement compiler (parsing, type checking, optimizations, etc.)
  - Discover parallelism (understand parallel patterns)
  - Emit parallel code for different hardware (optimize for low-level architectural details)
  - Handle synchronization, multiple address spaces, etc.

- Need a DSL infrastructure
  - Embed DSLs in a common host language
  - Provide building blocks for common DSL compiler & runtime functionality
Delite Overview

Domain Specific Languages

- Data Analytics (OptiQL)
- Physics (Liszt)
- Machine Learning (OptiML)

Delite: DSL Infrastructure

- Domain Embedding Language (Scala)
  - Staged Execution
- Delite Compiler
  - Parallel Patterns
  - Static Optimizations
  - Heterogeneous Code Generation
- Delite Runtime
  - Walk-time Optimizations
  - Locality-aware Scheduling

Heterogeneous Hardware

- SMP
- GPU
DSL Intermediate Representation (IR)

Domain User Interface

Domain Analysis & Opt.

Application

M1 = M2 + M3
V1 = exp(V2)
s = sum(M)
C2 = sort(C1)

Domain Ops

Matrix Plus
Vector Exp
Matrix Sum
Collection Quicksort

Domain User

DSL Author

Domain User Interface

Domain Analysis & Opt.
Building an IR

- OptiML: A DSL for machine learning
  - Built using Delite
  - Supports linear algebra (Matrix/Vector) operations

```scala
// a, b, c, d : Matrix
val x = a * b + c * d

def infix_+(a: Matrix, b: Matrix) =
  new MatrixPlus(a,b)

def infix_*(a: Matrix, b: Matrix) =
  new MatrixTimes(a,b)
```

- DSL methods build IR as program runs
DSL Optimizations

- DSL developer defines how DSL operations create IR nodes
- Specialize implementation of operation for each *occurrence* by pattern matching on the IR
- This technique can be used to control merely what to add to IR or to perform IR rewrites
  - Use this to apply linear algebra simplification rules

\[ AB + AC \Rightarrow A(B+C) \]
A straightforward translation of the Gaussian Discriminant Analysis (GDA) algorithm from the mathematical description produces the following code:

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

A much more efficient implementation recognizes that

$$
\sum_{i=0}^{n} \overrightarrow{x_i} \overrightarrow{y_i} \rightarrow \sum_{i=0}^{n} X(:,i) \ast Y(i,:) = X \ast Y
$$

Transformed code was 20.4x faster with 1 thread and 48.3x faster with 8 threads.
Delite DSL Framework

- Building a new DSL
  - Design the language (syntax, operations, abstractions, etc.)
  - Implement compiler
    - Domain-specific analysis and optimization
    - Lexing, parsing, type-checking, generic optimizations
  - Discover parallelism (understand parallel patterns)
  - Emit parallel code for different hardware (optimize for low-level architectural details)
  - Handle synchronization, multiple address spaces, etc.
Delite Ops

- Encode known parallel execution patterns
  - Map, filter, reduce, ...
  - Bulk-synchronous foreach
  - Divide & conquer

- Delite provides implementations of these patterns for multiple hardware targets
  - e.g., multi-core, GPU

- DSL author maps each domain operation to the appropriate pattern
  - Delite handles parallel optimization, code generation, and execution for all DSLs
Multiview Delite IR

Domain User Interface

Domain Analysis & Opt.

Parallelism Analysis & Opt.

Code Generation

DSL User

Application

M1 = M2 + M3
V1 = exp(V2)
s = sum(M)
C2 = sort(C1)

DSL Author

Domain Ops

Matrix Plus
Vector Exp
Matrix Sum
Collection Quicksort

Delite

Delite Ops

ZipWith
Map
Reduce
Divide & Conquer
Delite Op Fusion

- Operates on all loop-based ops
- Reduces op overhead and improves locality
  - Elimination of temporary data structures
  - Merging loop bodies may enable further optimizations
- Fuse both dependent and side-by-side operations
  - Fused ops can have multiple inputs & outputs
- Algorithm: fuse two loops if
  - $\text{size}(\text{loop1}) == \text{size}(\text{loop2})$
  - No mutual dependencies (which aren’t removed by fusing)
Downsampling in OptiML

![Bar chart showing normalized execution time for different numbers of processors and different configurations of OptiML and C++]

- **Normalized Execution Time**
- **Processors**: 1, 2, 4, 8
- **OptiML Fusing**
- **OptiML No Fusing**
- **C++**
Multiview Delite IR

Application

M1 = M2 + M3
V1 = exp(V2)
s = sum(M)
C2 = sort(C1)

Domain Ops

Matrix Plus
Vector Exp
Matrix Sum
Collection Quicksort

Delite Ops

ZipWith
Map
Reduce
Divide & Conquer

Domain User Interface

Domain Analysis & Opt.

Parallelism Analysis & Opt.

Code Generation

Generic Analysis & Opt.

DSL User

DSL Author

Delite

Delite

Generic Op
Generic IR

- Optimizations
  - Common subexpression elimination (CSE)
  - Dead code elimination (DCE)
  - Constant folding
  - Code motion (e.g., loop hoisting)

- Side effects and alias tracking

- All performed at the granularity of DSL operations
  - e.g., MatrixMultiply
Delite DSL Compiler Infrastructure

- **Liszt program**
  - Scala Embedding Framework
  - Intermediate Representation (IR)
    - Base IR
    - Generic Analysis & Opt.
  - Code Generation
    - Delite Execution Graph

- **OptiML program**
  - Delite Parallelism Framework
  - Delite IR
    - Parallelism Analysis, Opt. & Mapping
  - Code Generation
    - Kernels (Scala, C, Cuda)
    - DSL Data Structures
Heterogeneous Code Generation

- Delite can have multiple registered target code generators (Scala, Cuda, ...)
  - Calls all generators for each Op to create kernels
  - Only 1 generator has to succeed

- Generates an *execution graph* that enumerates all Delite Ops in the program
  - Encodes parallelism within the application
  - Contains all the information the Delite Runtime requires to execute the program
    - Op dependencies, supported targets, etc.
Delite Runtime

Machine Inputs
- Local System
  - SMP
  - GPU

Application Inputs
- Delite Execution Graph
- Kernels (Scala, C, Cuda)
- DSL Data Structures

Walk-Time
- Scheduler
- Code Generator
  - JIT Kernel Fusion, Specialization, Synchronization

Partial schedules, Fused, specialized kernels

Execution-Time
- Schedule Dispatch, Memory Management, Lazy Data Transfers
Schedule & Kernel Compilation

- Compile execution graph to executables for each resource after scheduling
  - Defer all synchronization to this point and optimize

- Kernels specialized based on number of processors allocated for it
  - e.g., specialize height of tree reduction

- Greatly reduces overhead compared to dynamic deferred execution model
  - Can have finer-grained Ops with less overhead
Benefits of Runtime Codegen

- GDA with 64 element input

![Bar chart showing normalized execution time for compilers and interpreters across different numbers of processors. The chart indicates that compiled code generally performs better than interpreted code, with normalized execution times for 1, 2, 4, and 8 processors being 1.00, 1.62, 2.30, and 3.21 for compilers and 0.99, 0.53, 0.62, and 0.49 for interpreters, respectively.]
GPU Management

- Cuda host thread launches kernels and automatically performs data transfers as required by schedule
  - Compiler provides helper functions to
    - Copy data structures between address spaces
    - pre-allocate outputs and temporaries
    - select the number of threads & thread blocks

- Provides device memory management for kernels
  - Perform liveness analysis to determine when op inputs and outputs are dead on the GPU
  - Runtime frees dead data when it experiences memory pressure
Cuda Code Generation

- With a library approach we can only launch pre-written kernels
- Code generation enables kernels containing user-defined functions and optimization opportunities
  - e.g., fuse operations into one kernel and keep intermediate results in registers

![Normalized Execution Time Graph]

<table>
<thead>
<tr>
<th></th>
<th>Library-Based</th>
<th>Delite</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBM</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>NB</td>
<td>2.3</td>
<td></td>
</tr>
<tr>
<td>GDA</td>
<td>5.5</td>
<td></td>
</tr>
</tbody>
</table>
Performance Results

- **Machine**
  - Two quad-core Nehalem 2.67 GHz processors
  - NVidia Tesla C2050 GPU

- **Application Versions**
  - OptiML + Delite
  - MATLAB
    - version 1: multi-core (parallelization using “parfor” construct and BLAS)
    - version 2: GPU
  - C++
    - used Armadillo linear algebra library for a sequential baseline
    - Algorithmically identical to OptiML version
OptiML vs. MATLAB vs. Armadillo (C++)

- **GDA**
  - Normalized Execution Time:
    - OptiML: 1.0, 1.6, 1.8, 1.9
    - Parallelized MATLAB: 1.2, 1.9, 2.1, 2.2
    - C++: 1.4, 1.1, 1.0, 1.0

- **Naive Bayes**
  - Normalized Execution Time:
    - OptiML: 1.0, 0.1, 0.2, 0.3
    - Parallelized MATLAB: 1.2, 1.9, 3.6, 5.8
    - C++: 1.4, 0.3, 0.2, 0.3

- **K-means**
  - Normalized Execution Time:
    - OptiML: 1.0, 1.2, 4.1, 7.1
    - Parallelized MATLAB: 0.3, 0.4, 0.4, 0.4
    - C++: 1.2, 0.3, 0.3, 0.3

- **SVM**
  - Normalized Execution Time:
    - OptiML: 1.0, 0.4, 3.1, 4.2
    - Parallelized MATLAB: 0.5, 0.2, 0.2, 0.2
    - C++: 0.8, 0.9, 1.4, 1.4

- **Linear Regression**
  - Normalized Execution Time:
    - OptiML: 1.0, 0.5, 0.9, 1.3
    - Parallelized MATLAB: 0.5, 1.4, 2.0, 2.3
    - C++: 1.7, 1.7, 1.1, 1.1

- **RBM**
  - Normalized Execution Time:
    - OptiML: 1.0, 1.0, 1.2, 1.9
    - Parallelized MATLAB: 0.6, 1.7, 2.7, 3.2
    - C++: 0.4, 0.7, 3.5, 4.7

- **CPU + GPU**
Conclusions

- DSLs can provide both productivity and performance on heterogeneous hardware
- Need to simplify the process of developing DSLs for parallelism
  - Delite provides a framework for creating heterogeneous parallel DSLs
  - Performs generic, parallel, and domain-specific optimizations in a single system
- Visit us at ppl.stanford.edu
  - Link to GitHub project
  - Related publications & projects