Accelerating Big Data Analytics with Domain Specific Languages

Kunle Olukotun
Cadence Design Systems Professor
Stanford University

36e Forum ORAP
CNRS, Paris
November, 5 2015
Research Goals

- Unleash full power of modern computing platforms on big data
  - Huge and growing amounts of data
  - Deliver the capability to mine, search and analyze this data in real time to enable discovery and decision making

- Make parallel application development practical for the masses (Joe/Jane the programmer)
  - Parallelism is not for the average programmer
  - Too difficult to find parallelism, to debug, and get good performance

- Parallel applications without parallel programming
Data Center Computing Platforms

- Multicore
- Multi-socket
- Graphics Processing Unit (GPU)
- Programmable Logic

> 1 TFLOPS

10s of cores

1000s of nodes

Cluster
Expert Parallel Programming Required

10s of cores
Multicore Muti-socket
Threads OpenMP

CUDA OpenCL
> 1 TFLOPS
Graphics Processing Unit (GPU)

Verilog VHDL
Programmable Logic

1000s of nodes
Cluster

MPI Map Reduce
MPI: Message Passing Interface
Expert Programmers $\Rightarrow$ Low Productivity
Too Few Expert Programmers

Image Filter OpenMP Assignment

Optimizations:
- Precomputing twiddle
- Not computing what isn’t part of the filter
- Transposing the matrix
- Using SSE

~3 orders of magnitude
Hypothesis

It is possible to write one program and run it efficiently on all these machines.
Domain Specific Languages (DSLs)

- Programming language with restricted expressiveness for a particular domain
- High-level, usually declarative, and deterministic
Big-Data Analytics Programming Challenge

Data Analytics Application
- Data Prep
- Data Transform
- Network Analysis
- Predictive Analytics

High-Performance Domain Specific Languages

Pthreads OpenMP
- Multicore

CUDA OpenCL
- GPU

MPI Map Reduce
- Cluster

Verilog VHDL
- FPGA
High Performance DSLs for Data Analytics

Applications
- Data Transformation
- Graph Analysis
- Prediction Recommendation

Domain Specific Languages
- Data Extraction: OptiWrangler
- Query Proc.: OptiQL
- Graph Alg.: OptiGraph
- Machine Learning: OptiML

Heterogeneous Hardware
- Multicore
- GPU
- FPGA
- Cluster
Delite: DSL Infrastructure

Applications
- Data Transformation
- Graph Analysis
- Prediction Recommendation

HP DSLs
- Data Extraction OptiWrangle
- Query Proc. OptiQL
- Graph Alg. OptiGraph
- Machine Learning OptiML

Delite DSL Framework
- DSL Compiler

Heterogeneous Hardware
- Multicore
- GPU
- FPGA
- Cluster
Delite Overview


Key elements

- DSLs embedded in Scala
- IR created using type-directed staging
- Domain specific optimization
- General parallelism and locality optimizations
- Optimized mapping to HW targets
Delite DSL Examples
OptiML: A DSL for Machine Learning

- Designed for Iterative Statistical Inference
  - e.g. SVMs, logistic regression, neural networks, etc.
  - Dense/sparse vectors and matrices, message-passing graphs, training/test sets

- Mostly Functional
  - Data manipulation with classic functional operators (map, filter) and ML-specific ones (sum, vector constructor, untilconverged)
  - Math with MATLAB-like syntax (a*b, chol(..), exp(..))
  - Mutation is explicit (.mutable) and last resort

- Implicitly parallel control structures
  - sum{...}, (0:end) {...}, gradient { ... }, untilconverged { ... }
  - Allow anonymous functions with restricted semantics to be passed as arguments of the control structures
K-means Clustering in OptiML

```scala
untilconverged(kMeans, tol){
  kMeans =>
  val clusters = samples.groupRowsBy {
    sample =>
      kMeans.mapRows(mean => dist(sample, mean)).minIndex
  }
  val newKmeans = clusters.map(e => e.sum / e.length)
  newKmeans
}
```

- No explicit map-reduce, no key-value pairs
- No distributed data structures (e.g. RDDs)
- No annotations for hardware design
- Efficient multicore and GPU execution
- Efficient cluster implementation
- Efficient FPGA hardware
OptiQL

- Data querying of in-memory collections
  - inspired by LINQ to Objects

- SQL-like declarative language
  - Key operations are query operators on the Table data structure
  - User-defined schema

- Use high-level semantic knowledge to implement query optimizer
  - Fusion eliminates temporary allocations
  - AoS $\Rightarrow$ SoA
  - Eliminate fields not used in query

- 5-100x performance advantage over LINQ
  - On 8 cores
// lineItems: Table[LineItem]
// Similar to Q1 of the TPC-H benchmark
val q = lineItems Where(_.l_shipdate <= Date(‘19981201’)).
  GroupBy(l => l.l_linestatus).
  Select(g => new Record {
    val lineStatus = g.key
    val sumQty = g.Sum(_.l_quantity)
    val sumDiscountedPrice =
      g.Sum(r => r.l_extendedprice*(1.0-r.l_discount))
    val avgPrice = g.Average(_.l_extendedprice)
    val countOrder = g.Count
  }) OrderBy(_.returnFlag) ThenBy(_.lineStatus)
OptiGraph

- A DSL for large-scale graph analysis based on Green-Marl
  - Green-Marl: A DSL for Easy and Efficient Graph Analysis (Hong et. al.), ASPLOS ’12

- Mostly functional DSL
  - Only explicit mutation, no explicit iteration

- Data structures
  - Graph (directed, undirected), node, edge,
  - Set of nodes, edges, neighbors, ...

- Graph traversals
  - Summation, Breadth-first traversal, ...
Example: Betweenness Centrality

Betweenness Centrality

- A measure that tells how ‘central’ a node is in the graph
- Used in social network analysis

Definition

- How many shortest paths are there between any two nodes going through this node.

\[ C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \]
OptiGraph Betweenness Centrality

1. val bc = sum(g.nodes){ s =>
2.  val sigma = g.inBfOrder(s) { (v,prev_sigma) => if(v==s) 1
3.  else sum(v.parents){ w => prev_sigma(w)}
4.  }
5.  val delta = g.inRevBfOrder(s){ (v,prev_delta) =>
6.    sum(v.children){ w =>
7.      sigma(v)/sigma(w)*(1+prev_delta(w))
8.    }
9.  }
10. delta
11. }
Delite Implementation
**DSL Goals**

BUILD **DOMAIN-SPECIFIC LANGUAGES** THAT ARE AS **PRODUCTIVE** AS LIBRARIES AND AS **PERFORMANT** AS PARALLEL CODE?

... for DSL **users**

high-level, composable, interactive, debuggable

... for DSL **authors**

easy to construct and extend
Delite: A Framework for High Performance DSLs

- Overall Approach: Generative Programming for “Abstraction without regret”
  - Embed compilers in Scala libraries
    - Scala does syntax and type checking
  - Use metaprogramming with LMS (type-directed staging) to build an intermediate representation (IR) of the user program
  - Optimize IR and map to multiple targets

- Goal: Make embedded DSL compilers easier to develop than stand alone DSLs
  - As easy as developing a library
Delite DSLs use LMS to build an IR from Scala application code:

```scala
val v1: Rep[Vector[Double]] = Vector.rand(1000)
val v2: Rep[Vector[Double]] = Vector.rand(1000)
val a: Rep[Vector[Double]] = v1+v2
println(a)
```

Method calls on `Rep[T]` instances construct nodes instead of evaluating. `Rep[T]` is an abstract type constructor.
Domain Specific Optimizations I

- Use domain-specific knowledge to make optimizations in a modular fashion
- Override IR node creation
  - Use Scala pattern matching
  - Construct Optimized IR nodes if possible
  - Construct default otherwise

\[
\begin{align*}
A & \rightarrow + \rightarrow A(B+C) \\
B & \rightarrow * \rightarrow AB + AC
\end{align*}
\]

A, B, C are large matrices

- Rewrite rules are simple, yet powerful optimization mechanism
Access to the full domain specific IR allows for application of much more complex optimizations

- Traversals for analysis
  - Schedules the IR and visit each node in order
  - DSL defines operation to execute at each node

- Transformers for optimizations
  - Combines traversals with rewrite rules
  - Transformed version is written as normal source code, then re-staged to an IR
Parallel Patterns: Delite Ops

- Parallel execution patterns
  - Functional: Map, FlatMap, ZipWith, Reduce, Filter, GroupBy, Sort, Join, union, intersection
  - Non-functional: Foreach, ForeachReduce, Sequential
  - Set of patterns can grow over time

- Provide high-level information about data access patterns and parallelism

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

- Delite provides implementations of these patterns for multiple hardware targets
  - High-level information creates straightforward and efficient implementations
  - Multi-core, GPU, clusters and FPGA
Parallel Patterns

Most data analytic computations can be expressed as parallel patterns on collections (e.g. sets, arrays, table)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>in map { e =&gt; e + 1 }</td>
</tr>
<tr>
<td>zipwith</td>
<td>inA zipWith(inB) { (eA,eB) =&gt; eA + eB }</td>
</tr>
<tr>
<td>foreach</td>
<td>inA foreach { e =&gt; if (e&gt;0) inB(e) = true }</td>
</tr>
<tr>
<td>filter</td>
<td>in filter { e =&gt; e &gt; 0}</td>
</tr>
<tr>
<td>reduce</td>
<td>in reduce { (e1,e2) =&gt; e1 + e2 }</td>
</tr>
<tr>
<td>groupby</td>
<td>in groupBy { e =&gt; e.id }</td>
</tr>
</tbody>
</table>

Other patterns: sort, intersection, union
Parallel Pattern Language (PPL)

- A data-parallel language that supports parallel patterns
- Example application: k-means

```scala
val clusters = samples groupBy { sample =>
  val dists = kMeans map { mean =>
    mean.zip(sample){ (a,b) => sq(a - b) } reduce { (a,b) => a + b }
  }
  Range(0, dists.length) reduce { (i,j) =>
    if (dists(i) < dists(j)) i else j
  }
}
val newKmeans = clusters map { e =>
  val sum = e reduce { (v1,v2) => v1.zip(v2){ (a,b) => a + b } } 
  val count = e map { v => 1 } reduce { (a,b) => a + b }
  sum map { a => a / count }
}
```
## Programming Model Comparison

<table>
<thead>
<tr>
<th>Programing Model</th>
<th>Rich Data Parallelism</th>
<th>Use Nested Parallelism</th>
<th>Multiple Collections</th>
<th>Random Reads</th>
<th>Hardware Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>MapReduce</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Cluster</td>
</tr>
<tr>
<td>DryadLINQ</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td>Cluster</td>
</tr>
<tr>
<td>Spark</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td>Cluster, Multicore</td>
</tr>
<tr>
<td>PowerGraph</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td>Cluster, Multicore</td>
</tr>
<tr>
<td>Dandelion</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td>Cluster, Multicore, GPU</td>
</tr>
<tr>
<td>Delite (PPL)</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>Cluster, Multicore, NUMA, GPU FPGA</td>
</tr>
</tbody>
</table>
Raising the Level of Abstraction for Accelerators

- A higher level programming model
  - Architecture independent
  - High-level enough to encode important semantic information
  - Efficient mapping to low-level programming models

Parallel Patterns

CUDA
nVIDIA TESLA

OpenCL
AMD APU

OpenMP / TBB
Intel Xeon Phi
Mapping Nested Parallel Patterns to GPUs

- Parallel patterns are often nested in applications
  - > 70% apps in Rodinia benchmark contain kernels with nested parallelism

- Efficiently mapping parallel patterns on GPUs becomes significantly more challenging when patterns are nested
  - Memory coalescing, divergence, dynamic allocations, ...
  - Large space of possible mappings

HyoukJoong Lee et al., “Locality-Aware Mapping of Nested Parallel Patterns on GPUs,” MICRO’14
2D mapping exposes more parallelism
2D mapping enables coalesced memory accesses
Distributed Heterogeneous Execution

- **Separate Memory Regions**
  - NUMA
  - Clusters
  - FPGAs

- **Partitioning Analysis**
  - Multidimensional arrays
  - Decide which data structures / parallel ops to partition across abstract memory regions

- **Nested Pattern Transformations**
  - Optimize patterns for distributed and heterogeneous architectures
Locality-Improving Transformations

Sometimes the optimal way of writing nesting parallel patterns is architecture dependent, e.g., Logistic Regression:

```scala
untilconverged(theta, tol) { theta =>
  (0 until x.numCols) map { j =>
    val gradient = sum(0, x.numRows) { i =>
      x(i,j)*(y(i) - hyp(theta, x(i)))
    }
    theta(j) + alpha*gradient
  }
}
```

```scala
untilconverged(theta, tol) { theta =>
  val gradientVec = sum(0, x.numRows) { i =>
    (0 until x.numCols) map { j =>
      x(i,j)*(y(i) - hyp(theta, x(i)))
    }
  }
  (0 until x.numCols) map { j =>
    theta(j) + alpha*gradientVec(j)
  }
}
```

GPU optimized

Cluster & multicore optimized
Parallel Patterns as Multiloops

- Lower data-parallel patterns to *multiloops* (non-affine) which contain one or more structured bodies (*generators*)

- **Collect**\( (\text{cond}) (f) \)**
  - Produces values from a function \( f \)
  - Creates a flat collection by conditionally \( (\text{cond}) \) appending the values (filter)
  - Covers Map, Zip, Filter, FlatMap, ...

- **BucketCollect**\( (\text{cond}) (\text{key}) (f) \)**
  - Produces (key, value) pairs from functions \( (\text{key}, f) \)
  - Creates a nested collection by appending values to the buffer for the key
  - Covers GroupBy, ...

- **Reduce**\( (\text{cond}) (f) (\text{red}) \)**
  - Produces values from a function \( f \)
  - Reduces those values using associative op \( r \)
  - Covers Reduce, Map-Reduce, Filter-Reduce, ...

- **BucketReduce**\( (\text{cond}) (\text{key}) (f) (\text{red}) \)**
  - Produces (key, value) pairs from functions \( (\text{key}, f) \)
  - Reduces values with like key using associative op \( r \)
  - Result is flat (one element per key)
  - Covers Distinct, Google’s *MapReduce*, ...

\[
\begin{align*}
\text{cond: Index} & \rightarrow \text{Boolean} \quad \text{key: Index} \rightarrow \text{K} \\
\text{f: Index} & \rightarrow \text{V} \quad \text{red: (V,V)} \rightarrow \text{V}
\end{align*}
\]
Rewriting Nested Patterns: GroupBy-Reduce Rule

Example: basic SQL query:
```java
lineItems.groupBy(item => item.type).map(group =>
    group.map(item => item.quantity).sum)
```

Issues:
- Intermediate output of `groupBy` can be very large compared to final output
- Limited parallelism over resulting groups

Rewrite rule fuses and flattens the nested reduction:
```java
A = BucketCollect_s(c)(k)(f_1) Collect_A(\_)(i => Reduce_A(i)(\_)(f_2)(r))
\rightarrow BucketReduce_s(c)(k)(f_2(f_1))(r)
```

Optimized version: single traversal without materializing intermediate collections
```java
bucketReduce(size = lineItems.size)
    (cond = i => true)
    (key = i => lineItems(i).type)
    (value = i => lineItems(i).quantity)
    (reduce = (a,b) => a + b)
```

Same rule can be applied to `groupBy` formulation of k-means, k-nearest neighbors, …
- Polyhedral analysis can’t do this optimization
Keys To Delite Performance

- Compilers exploit high-level semantics (domain specific opt.)
  - MatrixTimesVector instead of
  ```
  for i <- 0 until m.numRows {
    for j <- 0 until m.numCols {
      ...
  }
  ```

- Staging programmatically removes abstraction
  - Method calls inlined, modularity structures (traits, classes) compiled away
  - Generated code is low-level and first order

- Parallel patterns as first class citizens in the IR
  - Easy for DSL authors to expose parallelism

- Data structures are represented in the IR
  - Struct wrappers, fields, are statically eliminated when possible

- Analysis and transformation to optimize across parallel patterns
  - Delite provides a large set for all DSLs
  - Each DSL can add its own
Scala

```scala
def apply(x388:Int,x423:Int,x389:Int,
x419:Array[Double],x431:Int,
x433:Array[Double]) {
  val x418 = x413 * x389
  val x912_zero = { 0 }
  val x912_zero_2 = {
    1.7976931348623157E308 }
  var x912 = x912_zero
  var x912_2 = x912_zero_2
  var x425 = 0
  while (x425 < x423) {
    val x430 = x425 * 1
    val x432 = x430 * x431
    val x916_zero = {
      0.0
    }
  }
}
```

CUDA

```cuda
__device__ int
dev_collect_x478_x478(int x423,int x389,DeliteArray<double> x419,int x431,DeliteArray<double> x433,int x413) {
  int x418 = x413 * x389;
  int x919 = 0;
  double x919_2 = 1.7976931348623157E308;
  int x425 = 0;
  while (x425 < x423) {
    int x430 = x425 * 1;
    int x432 = x430 * x431;
    double x923 = 0.0;
    int x450 = 0;
    ...
  }
}
```

Abstraction without regret:
Eliminate higher-order abstractions in generated code
Delite DSL Performance
OptiML and OptiQL on Cluster

20 node Amazon cluster
Q1: TPC-H 5GB dataset; GDA: 17GB dataset
OptiML on Heterogeneous Cluster

4 node local cluster: 3.4 GB dataset
val \( pr = \) untilConverged(0,threshold){ oldPR =>
  g.mapNodes{ n =>
    ((1.0-damp)/g.numNodes) + damp*sum(n.inNbrs){ w =>
      oldPr(w)/w.outDegree
    }
  }
}((curPR,oldPR) => sum(abs(curPr-oldPr)))
Multi-socket NUMA Performance

1–48 threads
4 sockets

<table>
<thead>
<tr>
<th></th>
<th>TPCHQ1</th>
<th>Gene</th>
<th>GDA</th>
<th>LogReg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td><img src="#" alt="Graph" /></td>
<td><img src="#" alt="Graph" /></td>
<td><img src="#" alt="Graph" /></td>
<td><img src="#" alt="Graph" /></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>5x</td>
<td>10x</td>
<td>10x</td>
<td>10x</td>
</tr>
<tr>
<td>24</td>
<td>11x</td>
<td>45x</td>
<td>10x</td>
<td>10x</td>
</tr>
<tr>
<td>48</td>
<td>30x</td>
<td>30x</td>
<td>10x</td>
<td>10x</td>
</tr>
</tbody>
</table>

- **TPCHQ1**: Speedup ranges from 0 to 30x.
- **Gene**: Speedup ranges from 0 to 45x.
- **GDA**: Speedup ranges from 0 to 10x.
- **LogReg**: Speedup ranges from 0 to 10x.

**Legend:**
- No Pin
- Pin
- NUMA
- Spark
- PowerGraph

**Note:** The graphs show the performance improvement (Speedup) for different workloads across varying thread counts (1–48) and with different socket configurations.
MSM Builder Using OptiML with Vijay Pande

Markov State Models (MSMs)
MSMs are a powerful means of modeling the structure and dynamics of molecular systems, like proteins.

![Diagram showing MSMbuilder Kinetic Clustering](image)

- **OptiML**: high prod, high perf
- **C++, x86 ASM**: low prod, high perf
- **Python**: high prod, low perf
# Re-Usable DSL Components

<table>
<thead>
<tr>
<th>DSL</th>
<th>Delite Ops</th>
<th>Generic Optimizations</th>
<th>Domain Specific Optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>OptiML</td>
<td>Map, ZipWith, Reduce, Foreach, GroupBy, Sort</td>
<td>CSE, DCE, code motion, fusion</td>
<td>linear algebra</td>
</tr>
<tr>
<td>OptiQL</td>
<td>Map, Reduce, Filter, Sort, GroupBy, intersection</td>
<td>CSE, DCE, DFE, SoA, code motion, fusion</td>
<td>query optimization</td>
</tr>
<tr>
<td>OptiGraph</td>
<td>Map, Reduce, Filter, GroupBy</td>
<td>CSE, DCE, code motion</td>
<td>scatter, gather (push, pull)</td>
</tr>
<tr>
<td>OptiMesh</td>
<td>ForeachReduce</td>
<td>CSE, DCE, code motion</td>
<td>stencil collection, coloring transform</td>
</tr>
<tr>
<td>OptiWrangler</td>
<td>Map, ZipWith, Reduce, Filter, Sort, GroupBy</td>
<td>CSE, DCE, code motion, fusion</td>
<td></td>
</tr>
</tbody>
</table>
Real Applications Span Multiple Domains

Data cleansing and querying

Analytics

Visualization

Image sources: ThinkStock
http://technaverbascripta.wordpress.com/2012/10/19/text-network-analysis-2-meaning-circulation-in-lolita/
http://bl.ocks.org/mbostock/3943967
DSL Composition

- DSLs that require restricted semantics to enable domain-specific transformations

- We use a three step process:
  - Independently stage each DSL block using scopes
  - Lower each IR to the common IR (Delite IR) and combine together
  - Optimize and code generate the final composed IR

- Scopes
  - A Scala construct for type-safe isolated lexical scoping
DSL Composability

- Data analytic application with a Twitter dataset
- Uses OptiQL, OptiGraph, and OptiML
DSL Development Should Be Even Simpler

- The Problem
  - Embedded DSLs are high-level and high performance for end users
  - Easier to develop than external DSLs, easier to use than low-level code
  - ...but they are still not as easy to develop or use as libraries

- Significant Boilerplate with Embedding:
  - Emulate DSL syntax
  - Isolate the DSL program from the DSL compiler
  - Extend parallel patterns
  - Implement new data structures

- Debugging is Painful
  - Compilation is slow
  - Interactive programming is hard
Forge: A Meta-DSL

- **Delite**
  - is a highly flexible compiler for high performance in heterogeneous environments
  - but it is too low-level for the average DSL developer

- **Raising the level of abstraction**
  - Idea: let’s apply DSL principles to DSL development
  - Let DSL authors specify what, not how

- **Forge: a meta dsl for dsl development**
  - A declarative language for specifying DSL behavior
  - Can generate multiple concrete implementations of a DSL
Forge Compilation Pipeline

- **Productivity**
  - Scala Library
    - REPL, IDE, Scaladoc

- **Performance**
  - Delite DSL
  - Optimization and Code Generation
  - Multicore, GPU, Cluster

- **Inputs**
  - External DSL code
  - DSL Specification

- **Outputs**
  - DSL Application
High Performance Data Analytics with Delite

Applications
- Data Transformation
- Graph Analysis
- Prediction Recommendation

Domain Specific Languages
- Data Extraction OptiWrangler
- Query Proc. OptiQL
- Graph Alg. OptiGraph
- Machine Learning OptiML

Delite DSL Framework
- DSL Compiler
- DSL Compiler
- DSL Compiler
- DSL Compiler

Heterogeneous Hardware
- Multicore
- GPU
- FPGA
- Cluster

✓ ✓ ✓
FPGAs in the Datacenter?

- FPGAs based accelerators
  - Recent commercial interest from Baidu, Microsoft, and Intel
  - Key advantage: Performance, Performance/Watt
  - Key disadvantage: lousy programming model

- Verilog and VHDL poor match for software developers
  - High quality designs

- High level synthesis (HLS) tools with C interface
  - Medium/low quality designs
  - Need architectural knowledge to build good accelerators
  - Not enough information in compiler IR to perform access pattern and data layout optimizations
  - Cannot synthesize complex data paths with nested parallelism
Optimized Approach to HW Generation

Key optimizations:
- Parallel pattern tiling to maximize on-chip data reuse
- Metapipelines to exploit nested parallelism

Generate MaxJ code
- Use Maxeler’s MaxCompiler to generate FPGA bitstream
Summary

- In the age of heterogeneous architectures
  - Power limited computing $\Rightarrow$ parallelism and accelerators

- Staged embedded DSLs can be productive
  - DSL users program at a high-level
  - Embedded DSLs with a common backend can efficiently compose
  - Staging and re-use makes language construction much simpler
  - Meta DSLs reduce the burden on DSL developers even further

- And high performance
  - Additional semantic information and restrictions enables compiler to do domain-specific optimization
  - Staging programmatically removes high-level, library abstraction
  - Parallel patterns enable aggressive parallelism and locality optimizations through transformations
  - Efficient mapping to heterogeneous architectures using parallel patterns
Colaborators & Funding

**Faculty**
- Pat Hanrahan
- Martin Odersky (EPFL)
- Chris Ré
- Tiark Rompf (Purdue/EPFL)

**PhD students**
- Chris Aberger
- Kevin Brown
- Hassan Chafi
- Zach DeVito
- Chris De Sa
- Nithin George (EPFL)
- David Koeplinger

**Funding**
- PPL: Oracle Labs, Nvidia, Intel, AMD, Huawei, SAP
- NSF
- DARPA
- HyoukJoong Lee
- Victoria Popic
- Raghu Prabhakar
- Aleksander Prokopec (EPFL)
- Vojin Jovanovic (EPFL)
- Vera Salvisberg (EPFL)
- Arvind Sujeeth
Big Data Analytics
In the Age of Accelerators

- Power
- Performance
- Productivity
- Portability

Accelerators (GPU, FPGA, ...)

High Performance DSLs (OptiML, OptiQL, ...)

Delite
Thank You!

- Questions?

- Delite repository: http://stanford-ppl.github.com/Delite

- Stanford Pervasive Parallelism Lab: http://ppl.stanford.edu