Delite: A Framework for Implementing Heterogeneous Parallel DSLs

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

Tiark Rompf, Martin Odersky
EPFL
Outline

- Why high performance DSLs?
- Examples of high performance DSLs
- Delite: reusable components for DSL compilers
- Delite: optimizing for heterogeneous architectures
- Performance results
Heterogeneous Parallel Programming

Only way to get high performance and performance/watt
Programmability Chasm

Applications

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

Parallel Programming Language

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

Hardware

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

Performance

Productivity

Generality
Successful Languages

Performance

Productivity

Generality

C/C++

Java

Python

Ruby
Domain Specific Languages

Performance (Heterogeneous Parallelism)

Domain Specific Languages

Productivity

Generality
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
Our Approach: Three Views

- Little embedded languages
  - Domain abstractions improve productivity
  - Domains provide specific knowledge

- Smart libraries
  - Libraries that can compile/optimize themselves
  - Optimizations cross library call boundaries
  - Optimizations exploit domain specific knowledge

- Smart compilers
  - Raise abstraction-level of compiler optimization
  - Load and stores \(\Rightarrow\) Data structures
  - Language statements \(\Rightarrow\) Algorithms
Outline

- Why high performance DSLs?
- Examples of high performance DSLs
  - Delite: reusable components for DSL compilers
  - Delite: optimizing for heterogeneous architectures
- Performance results
OptiML

OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning, ICML 2011

- Provides a familiar (MATLAB-like) language and API for writing ML applications
  - Ex. `val c = a * b` (a, b are Matrix[Double])

- Implicitly parallel data structures
  - Base types: Vector[T], Matrix[T], Graph[V,E], Stream[T]
  - Subtypes: TrainingSet, IndexVector, Image, ...

- Implicitly parallel control structures
  - `sum{...}`, `(0::end) {...}`, `gradient { ... }`, `untilconverged { ... }
  - Arguments to control structures are anonymous functions with restricted semantics
OptiML: k-means Clustering

untilconverged(mu, tol){
    mu =>
    // calculate distances to current centroids
    val c = (0::m){i =>
        val allDistances = mu mapRows { centroid =>
            dist(x(i), centroid)
        }
        allDistances.minIndex
    }

    // move each cluster centroid to the
    // mean of the points assigned to it
    val newMu = (0::k,*){ i =>
        val (weightedpoints, points) = sum(0,m) { j =>
            if (c(i) == j) (x(i),1)
        }
        val d = if (points == 0) 1 else points
                weightedpoints / d
    }
    newMu
}
**k-means Generated Code**

**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
        ...
    }
    Abstraction without regret: Eliminate higher-order abstractions in generated code
```
OptiQL

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

- SQL-like declarative language

- Use high-level semantic knowledge to implement query optimizer
OptiQL: TPCH-Q1

// 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 Result {
    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

- Directed and undirected graphs, nodes, edges

- Collections for node/edge storage
  - Set, sequence, order

- Deferred assignment and parallel reductions with bulk synchronous consistency
OptiGraph: PageRank

```
for(t <- G.Nodes) {
    val rank = ((1.0 - d) / N) +
               d * Sum(t.InNbrs){w => PR(w) / w.OutDegree}
    PR <= (t,rank)
    diff += abs(rank - PR(t))
}
```

Implicitly parallel iteration

Deferred assignment and scalar reduction

Writes become visible after the loop completes
OptiMesh

- DSL for solving discretized PDEs on a 3D mesh
  - Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers (DeVito, et. al.), SC 11

- Topological mesh elements
  - vertex, edge, face, cell
  - Functions: e.g., head(edge)

- Data storage at mesh elements
  - Scalars, fixed-width vectors

- Automatically infer PDE’s stencil
  - Avoid write conflicts by coloring loops
val Flux = Field[Vertex,Double](0.0)

for(edge <- edges(mesh)) {
  val flux = flux_calc(edge)
  val v0 = head(edge)
  val v1 = tail(edge)
  Flux(v0) += flux
  Flux(v1) -= flux
}
Outline

- Why high performance DSLs?
- Examples of high performance DSLs
  - Delite: reusable components for DSL compilers
  - Delite: optimizing for heterogeneous architectures
- Performance results
Building a Language is Difficult

- 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
Delite: A DSL Framework

- DSLs embedded in Scala to leverage Scala compiler’s front-end (parsing, type-checking)
  - Generates IR from DSL application

- Reusable IR shares common optimizations across DSLs
  - Extends parallel patterns

- Parallel patterns implemented efficiently on heterogeneous architectures
  - Delite Ops
Matrix Example

```scala
import Matrix

trait TestMatrix {
  def example(a: Matrix, b: Matrix, c: Matrix, d: Matrix) = {
    val x = a*b + a*c
    val y = a*c + a*d
    println(x+y)
  }
}

- How do we construct an IR of this?
```
Building an IR

- DSL methods build IR as program runs

```python
def infix_+(a: Matrix, b: Matrix) = 
    
def infix_*(a: Matrix, b: Matrix) = 
    
A \Rightarrow Matrix
\times Matrix
\times Matrix
\times Matrix
Plus
```
DSL Intermediate Representation (IR)

**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 Interface**

**Domain Analysis & Opt.**

**DSL User**

**DSL Author**
DSL Optimizations

- Override IR node creation for each occurrence by pattern matching
  - Construct optimized IR nodes if possible
  - Construct default otherwise

- Rewrite rules are simple, yet powerful optimization mechanism
- Domain specific IR allows for much more complex optimizations
OptiML Linear Algebra Rewrites

- A straightforward translation of the Gaussian Discriminant Analysis (GDA) algorithm from the mathematical description produces the following code:

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

- A much more efficient implementation recognizes that

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

- Transformed code was 20.4x faster with 1 thread and 48.3x faster with 8 threads.
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

Delite

DSL User
DSL Author

Domain User
Interface

Domain Analysis & Opt.

Parallelism Analysis & Opt.

Code Generation

Application

Domain Ops

Delite Ops

C2 = sort(C1)

M1 = M2 + M3

V1 = exp(V2)

s = sum(M)
**Delite Ops**

- Encode known parallel execution patterns
  - Map, FlatMap, Reduce, ZipWith, Foreach, Filter, Hash (GroupBy), Join, Sort

- 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
  - e.g., multi-core, GPU
Delite Op Fusing

- Fuse Parallel Ops
  - Fuse producer-consumer operations
    - Communicate through registers
    - Eliminate temporary data structures
  - Fuse sibling operations
    - Produce all outputs with a single pass over the input

- Reduces Op overhead and improves locality
  - Enables further optimizations

- Allows you to write application in a clear functional style without paying for the overhead
Downsampling in OptiML

![Graph showing normalized execution time for different processor counts with C++, OptiML Fusing, and OptiML No Fusing categories.](image)
Multiview Delite IR

**Domain User Interface**

**Domain Analysis & Opt.**
- Matrix Plus
- Vector Exp
- Matrix Sum
- Collection Quicksort

**Parallelism Analysis & Opt.**

**Code Generation**

**Generic Analysis & Opt.**

**Application**
- \( M_1 = M_2 + M_3 \)
- \( V_1 = \exp(V_2) \)
- \( s = \text{sum}(M) \)
- \( C_2 = \text{sort}(C_1) \)

**Domain Ops**

**Delite Ops**
- ZipWith
- Map
- Reduce
- Divide & Conquer

**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
  - DSL author annotates operations with type of effect

- All performed at the granularity of DSL operations
  - e.g., MatrixMultiply
# Re-Usable DSL Components

<table>
<thead>
<tr>
<th>DSL</th>
<th>Delite Ops</th>
<th>Generic Optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>OptiML</td>
<td>Map, ZipWith, Reduce, Foreach, Hash, Sort</td>
<td>CSE, DCE, code motion, fusion</td>
</tr>
<tr>
<td>OptiQL</td>
<td>Map, Reduce, Filter, Sort, Hash, Join</td>
<td>CSE, DCE, code motion, fusion</td>
</tr>
<tr>
<td>OptiGraph</td>
<td>ForeachReduce, Map, Reduce, Filter</td>
<td>CSE, DCE, code motion</td>
</tr>
<tr>
<td>OptiMesh</td>
<td>ForeachReduce</td>
<td>CSE, DCE, code motion</td>
</tr>
</tbody>
</table>
Outline

- Why high performance DSLs?
- Examples of high performance DSLs
- Delite: reusable components for DSL compilers
- Delite: optimizing for heterogeneous architectures
- Performance results
Don’t want DSL developers deal with hardware details (CUDA code generation, thread/memory management)
Overview of the DSL Compilation and Execution

```
// 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))
  }
```

---

**DSL application**

**Delite Compiler**

**Delite Runtime**

(Walk-time analysis)
Outline

- Delite: optimizing for heterogeneous architectures
  - Compile-time DeliteOp code generation
  - Walk-time device-specific optimization
Overview of the DSL Compilation and Execution

DSL application

Delite Compiler

Delite Runtime
Heterogeneous Code Generation

- Delite supports adding new code generators for any target language
  - e.g., Scala, Cuda, OpenCL, C, LLVM

- Delite calls every generator for each Op to create kernels
  - DSL chooses which code generators to use
  - Only 1 generator has to succeed

- Delite also generates an *execution graph* that enumerates all Delite Ops in the program
  - Encodes task & data parallelism within the application
  - Contains all the information the Delite Runtime requires to execute the program
    - Op dependencies, supported targets, etc.
GPU Code Generation

Domain User Interface

Domain Analysis & Opt.

Parallelism Analysis & Opt.

Code Generation

Generic Analysis & Opt.

Application

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

Domain Ops

Matrix Plus
Vector Exp
Matrix Sum
Collection Quicksort

Delite Ops

ZipWith
Map
Reduce
Divide & Conquer

Op

Generic Op

DSL User

DSL Author

Delite

Delite
DeliteOp Code Generation

- For every DeliteOp, CPU kernel code (Scala) is generated.
- GPU kernel code (Cuda / OpenCL) will be generated only for data-parallel DeliteOps (e.g., DeliteOpForeach).

```scala
while (iter < 4) {
  val factor = deltat / (5.0 - iter)  // non data-parallel kernel

  for (f <- faces(mesh)) {
    Phi(f) = PhiOld(f) + factor/face_area(f)*Flux(f)  // data-parallel kernel (foreach type)
  }
}
```
DeliteOp Code Generation

- For every DeliteOp, CPU kernel code (Scala) is generated
- GPU kernel code (Cuda / OpenCL) will be generated only for data-parallel DeliteOps (e.g., DeliteOpForeach)

```scala
while (iter < 4) {
    val factor = deltat / (5.0 - iter) // non data-parallel kernel
    for (f <- faces(mesh)) {
        Phi(f) = PhiOld(f) + factor/faces_area(f)*Flux(f) // data-parallel kernel (foreach type)
    }
}
```

```c
__global__ void kernel_x4974(int x34, MeshSet<Face> x33, Field<double> x7, double x757, Field<double> x6) {
    int idxX = blockIdx.x*blockDim.x + threadIdx.x;
    int x1548 = idxX;
    if( x1548 < x34 ) {
        Face x1549 = x33.dcApply(x1548);
        int x1550 = internal(x1549);
        double x1553 = x757 / x1552;
        ...
        double x1556 = x1551 + x1555;
        x6.update(x1550,x1556);
    }
}
```

DSL Application (OptiMesh)

- For every DeliteOp, CPU kernel code (Scala) is generated
- GPU kernel code (Cuda / OpenCL) will be generated only for data-parallel DeliteOps (e.g., DeliteOpForeach)

Assignment of elements to threads (Mapping by the compiler)
DeliteOp Code Generation

- For every DeliteOp, CPU kernel code (Scala) is generated
- GPU kernel code (Cuda/OpenCL) will be generated only for data-parallel DeliteOps (e.g., DeliteOpForeach)

```scala
while (iter < 4) {
    val factor = deltat / (5.0 - iter) // non data-parallel kernel
    for (f <- faces(mesh)) {
        Phi(f) = PhiOld(f) + factor / face_area(f) * Flux(f) // data-parallel kernel (foreach type)
    }
}
```

**DSL Application (OptiMesh)**

Just function body for each element (Defer the mapping to walk-time)

```c
__device__ void kernel_x4974(MeshSet<Face> x33, int x1548, ...) {
    Face x1549 = x33.dcApply(x1548);
    int x1550 = internal(x1549);
    double x1553 = x757 / x1552;
    ...
    double x1555 = x1553 * x1554;
    double x1556 = x1551 + x1555;
    x6.update(x1550, x1556);
}
```
Data Structure Generation

- Datastructures of the DSL for each target
  - e.g.) vector / matrix data types for Scala, Cuda, OpenCL, etc.

```scala
class FloatVector {
  var data: Array[Float]
  var length: Int
  var isRow: Boolean

  // Instance Methods
  ...
}
```

```c++
class Vector<Float> {
  float *data;
  int length;
  bool isRow;

  // Instance Methods
  ...
};
```

- Helper Functions
  - Memory transfer functions (Host ↔ Device) for each data type of the DSL
Data Structure Generation

- Datastructures of the DSL for each target
  - e.g.) vector / matrix data types for Scala, Cuda, OpenCL, etc.

```java
class FloatVector {
  var data: Array[Float]
  var length: Int
  var isRow: Boolean
  // Instance Methods
  ...
}

class Vector<Float> {
  float *data;
  int length;
  bool isRow;
  // Instance Methods
  ...
}
```

- Helper Functions
  - Memory transfer functions (Host ⇔ Device) for each data type of the DSL

```java
void copyMutableInputDtoH(JNIEnv *env, jobject obj, IndexVector *x158) {
  jclass cls = env->GetObjectClass(obj);
  jmethodID mid_data = env->GetMethodID(cls, "data", "()[I");
  jintArray data = (jintArray)(env->CallObjectMethod(obj, mid_data));
  jint *dataPtr = (jint *)env->GetPrimitiveArrayCritical(data, 0);
  jint *hostPtr;
  DeliteCudaMallocHost((void**)&hostPtr, x158->length * sizeof(int));
  int *devPtr = x158->data;
  DeliteCudaMemcpyDtoHAsync(hostPtr, x158->data, x158->length * sizeof(int));
  memcpy(dataPtr, hostPtr, x158->length * sizeof(int));
  env->ReleasePrimitiveArrayCritical(data, dataPtr, 0);
}
```

Transfer Function

Scala Object

Cuda(C++) Object
Overview of the DSL Compilation and Execution

```java
// 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**

**Delite Runtime**
(Walk-time analysis)
Walk-time Mapping to GPU

- Map Operation
- Reduce Operation
- Filter Operation
- Fused Operation
1. Map Operation

val vec2 = vec1 + 2.0f

__device__ float dev_collect_x33(DenseVector<float> x19, int x23) {
    float x30 = x19.apply(x23);
    return (x30 + 2.0);
}

__global__ void MultiLoop_GPU_Array_x33MapReduce(DenseVector<float> x33, int x22, DenseVector<float> x19) {
    int idxX = blockIdx.x * blockDim.x + threadIdx.x;
    if (idxX < x22) {
        float collect_x33 = dev_collect_x33(x19, idxX);
        x33.dcUpdate(idxX, collect_x33);
    }
}

MultiLoop_GPU_Array_x33MapReduce<<<dim3((1 +((x22 - 1)/1024)),1,1),dim3(1024,1,1),0,kernelStream>>>(**x33,x22,*x19);

DSL Program

Generated at walk-time

Generated by Compiler

Runtime calls the generated GPU kernel
2. Reduce Operation

- M input elements reduction on GPU
  - Assign N elements for each thread-block
  - Each thread-block reduces to a single element using shared memory
    - Ends up with M/N elements
  - Cannot synchronize across thread-blocks
    - Need to launch a separate kernel to reduce M/N elements

```
val vec2 = vec1.sum
```
2. Reduce Operation

```cpp
val vec2 = vec1.sum

__device__ float dev_reduce_x29(float x26, float x27) {
    float x28 = x26 + x27;
    return x28;
}
```

```
__global__ void MultiLoop_GPU_Array_x29MapReduce(float *temp_x29, float *temp_x29_2, int x22, DenseVector<float> x19) {
    int idxX = blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ float smem_x29[1024];
    smem_x29[threadIdx.x] = (idxX < x22) ? dev_collect_x29(x19, idxX) : 0;

    __syncthreads();

    for(unsigned int s=1; s<blockDim.x; s*=2) {
        if((idxX%(2*s))==0)
            smem_x29[threadIdx.x] = dev_reduce_x29(smem_x29[threadIdx.x], smem_x29[threadIdx.x+s]);
        __syncthreads();
    }

    if(threadIdx.x==0) temp_x29[blockIdx.x] = smem_x29[0];
}
```

Use of GPU shared memory

Generated by compiler

Generated at walk-time
3. Filter Operation

```scala
val vec2 = vec1 filter (e => e > 0)
```

__device__ bool dev_cond_x34(DenseVector<float> x19, int x23) {
    float x31 = x19.apply(x23);
    bool x32 = x31 > 0;
    return x32;
}

```

Input

<table>
<thead>
<tr>
<th>3</th>
<th>2</th>
<th>-1</th>
<th>5</th>
<th>-4</th>
<th>-1</th>
<th>-1</th>
<th>-1</th>
<th>-1</th>
<th>-9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

== allocate output memory ==

```

Generated by compiler

```java
Generated at walk-time
```

Condition bit-vector

Scan (prefix sum)

output
4. Fusing Primitive Operations

- All fused into a single loop by compiler
- Efficiently mapping it to a GPU device is enabled by
  - Each output of fused loop is tagged with original semantics (e.g., map, reduce, filter)

```scala
val vector_1 = (0::numElem) { e => func_1(e) }    // map
val s = vector_1.sum                                             // reduce

val vector_2 = (0::numElem) { e => func_2(e) }    // map
val c = vector_2.count{ e => e > 0}                                // filter, reduce
```
Overview of the DSL Compilation and Execution

DSL application

Delite Compiler

Delite Runtime
(Walk-time analysis)
Delite Runtime Execution Plan

- A series of kernel launches with synchronization & memory management
  - Ingredients from compiler: kernels, DEG, helper functions
  - Generates a static execution plan for each target resource

```plaintext
main {
  val x1 = CPU_kernel_x1()
  val x2 = GPU_kernel_x2(x1)
  val x3 = GPU_kernel_x3(x2)
  val x4 = CPU_kernel_x4(x3)
}
```

```plaintext
void GPU_Exec_Plan {
  wait_for_x1(); // sync
  copy_HtoD_x1(); // data transfer
  alloc_x2(); // mem alloc
  x2 = exec_kernel_x2(x1); // kernel call
  set_x1_freeable(); // mem free
  alloc_x3(); // mem alloc
  x3 = exec_kernel_x3(x2); // kernel call
  set_x2_freeable(); // mem free
  copy_DtoH_x3(); // data transfer
  set_ready_x3(); // sync
}
```
Performance Results

- DSL Implementations

<table>
<thead>
<tr>
<th>DSL</th>
<th>Domain</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>OptiML</td>
<td>Machine Learning</td>
<td>MATLAB, C++</td>
</tr>
<tr>
<td>OptiMesh</td>
<td>Mesh-based Solver</td>
<td>Liszt</td>
</tr>
<tr>
<td>OptiQL</td>
<td>Database Query</td>
<td>LINQ</td>
</tr>
<tr>
<td>OptiGraph</td>
<td>Graph Analysis</td>
<td>Green Marl</td>
</tr>
</tbody>
</table>

- Machine Specification
  - CPU: Two quad-core Nehalem 2.67 GHz
  - GPU: NVidia Tesla C2050
OptiML

Normalized Execution Time

K-means

<table>
<thead>
<tr>
<th>CPU</th>
<th>OptiML</th>
<th>Parallelized MATLAB</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>1.6</td>
<td>0.3</td>
</tr>
<tr>
<td>2</td>
<td>1.9</td>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>4</td>
<td>3.6</td>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>8</td>
<td>5.1</td>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>CPU + GPU</td>
<td>10.7</td>
<td>0.3</td>
<td></td>
</tr>
</tbody>
</table>

RBM

<table>
<thead>
<tr>
<th>CPU</th>
<th>OptiML</th>
<th>Parallelized MATLAB</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>1.8</td>
<td>0.9</td>
</tr>
<tr>
<td>2</td>
<td>1.8</td>
<td>1.8</td>
<td>1.1</td>
</tr>
<tr>
<td>4</td>
<td>3.1</td>
<td>3.2</td>
<td>3.2</td>
</tr>
<tr>
<td>8</td>
<td>3.1</td>
<td>4.5</td>
<td>4.5</td>
</tr>
<tr>
<td>CPU + GPU</td>
<td>7.6</td>
<td>8.0</td>
<td></td>
</tr>
</tbody>
</table>
OptiQL: TPCH-Q1

Normalized Execution Time

LINQ
OptiQL Baseline
OptiQL Optimized

1 core

Normalized Execution Time

8 cores

0.6
1.0
5.0
1.4
3.0
25.0
OptiGraph: PageRank

Normalized Execution Time

1P  2P  4P  8P
PageRank (100k nodes x 800k edges)

1.7  1.7  1.7

PageRank (8M nodes x 64M edges)

1P  2P  4P  8P
OptiGraph  1.3  2.4  4.8
Green Marl  2.1  3.9  4.3
OptiMesh

Normalized Execution Time

0.035

Shallow Water

Scalar Convection

OptiMesh GPU

Liszt GPU

Liszt CPU

0.33
**Summary**

- DSLs can be used for both high productivity and high performance
  - Abstraction without regret

- Delite provides re-usable components for creating new parallel DSLs
  - Common IR with optimizations
  - Parallel patterns
  - Heterogeneous code generation

- Performance results comparable with hand-optimized implementations across multiple domains
Thank You!

- Questions?

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

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

- Constructs of the embedding language can be overridden by the DSL:

  ```
  if (cond) thenBlock else elseBlock
  ```

  maps to

  ```
  ifThenElse(cond, thenBlock, elseBlock)
  ```

- DSL developer can control the meaning of conditionals by providing overloaded variants specialized to DSL types
Data Structures

- Delite manages data structures for DSLs
  - everything is a Struct/Record, specified programmatically
  - instantiation and field access lifted into IR

- We auto-generate the back-end implementation to different platforms
  - Also generate transfer functions between devices

- Delite reasons about Struct usage
  - Struct wrapper eliminated in generated code; just passes around required fields
  - Unused fields eliminated all together
  - Performs AoS -> SoA optimization transparently