A DSL for Physical Simulation on GPUs and CPUs

Gilbert Bernstein
Chinmayee Shah
Crystal Lemire
Zach DeVito
Matthew Fisher
Philip Levis
Pat Hanrahan

CPU  GPU
Porting Code is Expensive

Simulation

CPU  GPU  Cluster
Porting Code is Expensive

Porting Code is Repetitive

Simulation

Sim 2
Sim 3
Sim 4

CPU
GPU
Cluster

Sim 1

CPU
GPU
Cluster
Languages Abstract Hardware

“Domain-Specific” Language

Sim 1
Sim 2
Sim 3
Sim 4

CPU
GPU
Cluster

Existing Languages in Graphics

GLSL
Pixar’s RenderMan
Halide
Darkroom
What’s tricky about designing languages for Simulation?

Simulations of Diverse Phenomena

images © Ron Fedkiw, Doug James, Chris Wojtan, Rahul Narain, Andrew Selle
Simulations Couple Phenomena

images © Ron Fedkiw, Doug James, Chris Wojtan, Rahul Narain, Andrew Selle

Simulations use Diverse Geometric Structures

images © Ron Fedkiw, Doug James, Chris Wojtan, Rahul Narain, Andrew Selle
Diverse Geometric Structures

- 2d Grid
- Particles
- Triangle Mesh
- Linked Chains
Diverse Geometric Structures

Existing Languages Rely on the Data Model
A relational model of data for large shared data banks

[Codd 1970]
Example

Code

```
import "ebb"
local L = require "ebblib"
```

Import the language into a Lua Script
import "ebb"
local L = require "ebblib"

local TetLib = require 'ebb.domains.TetLib'
local dragon = TetLib.LoadTetmesh('dragon.veg')

Load the dragon mesh using the Tetrahedral Mesh Library

load "ebb"
local L = require "ebblib"

local TetLib = require 'ebb.domains.TetLib'
local dragon = TetLib.LoadTetmesh('dragon.veg')

local dt = L.Constant(L.double, 0.0002)
local K = L.Constant(L.double, 4.0)
local maxvel = L.Global(L.double, 0)

Setup Simulation
Constant Values
& Global Values
import "ebb"
local L = require "ebblib"
local TetLib = require 'ebb.domains.TetLib'
local dragon = TetLib.LoadTetmesh('dragon.veg')
local dt = L.Constant(L.double, 0.0002)
local K = L.Constant(L.double, 4.0)
local maxvel = L.Global(L.double, 0)

dragon.vertices:NewField('vel', L.vec3d):Load({0,0,0})
dragon.vertices:NewField('nxt_vel', L.vec3d):Load(...)
dragon.vertices:NewField('nxt_pos', L.vec3d):Load(...)
dragon.edges:NewField('rest_len', L.double):Load(0)

local ebb init_rest_len ( e : dragon.edges )
  var diff = e.head.pos - e.tail.pos
  e.rest_len = L.length(diff)
end

declare & initialize fields of data

local ebb init_rest_len ( e : dragon.edges )
  var diff = e.head.pos - e.tail.pos
  e.rest_len = L.length(diff)
end

ebb functions define per-element computations
import "ebb"
local L = require "ebblib"

local TetLib = require 'ebb.domains.TetLib'
local dragon = TetLib.LoadTetmesh('dragon.veg')

local dt = L.Constant(L.double, 0.0002)
llocal K = L.Constant(L.double, 4.0)
llocal maxvel = L.Global(L.double, 0)

dragon.vertices:NewField('vel', L.vec3d):Load({0, 0, 0})
dragon.vertices:NewField('nxt_vel', L.vec3d):Load(...) 
dragon.vertices:NewField('nxt_pos', L.vec3d):Load(...) 
dragon.edges:NewField('rest_len', L.double):Load(0)

donlocal ebb init_rest_len ( e : dragon.edges ) 
  var diff = e.head.pos - e.tail.pos
  e.rest_len = L.length(diff)
end

dragon.edges:foreach(init_rest_len)

-- initialize vel and acc --

Ebb understands how fields are accessed

Functions launch over sets of elements
import "ebb"
local L = require "ebblib"

local Tetlib = require 'ebb.domains.Tetlib'
local dragon = Tetlib.LoadTetmesh('dragon.veg')

local dt = L.Constant(L.double, 0.0002)
local K = L.Constant(L.double, 4.0)
local maxvel = L.Global(L.double, 0)

dragon.vertices:NewField('vel', L.vec3d):Load({0, 0, 0})
dragon.vertices:NewField('nxt_vel', L.vec3d):Load(...)
dragon.vertices:NewField('nxt_pos', L.vec3d):Load(...)
dragon.edges:NewField('rest_len', L.double):Load(0)

local ebb init_rest_len ( e : dragon.edges )
  var diff = e.head.pos - e.tail.pos
  e.rest_len = L.length(diff)
end

dragon.edges:foreach(init_rest_len)

-- initialize vel and acc --

local ebb compute_max_vel ( v : dragon.vertices )
  maxvel max = L.length(v.vel)
end

local ebb compute_acc ( v : dragon.vertices )

  var force = { 0.0, 0.0, 0.0 }

  -- Spring Force
  var mass = 0.0
  for e in v.edges do
    var diff = e.head.pos - v.pos
    var scale = (e.rest_len / L.length(diff)) - 1.0
    mass += e.rest_len
    force -= K * scale * diff
  end

  v.nxt_pos = v.pos + dt * v.vel + 0.5*dt*dt * force/mass
  v.nxt_vel = v.vel + dt * force/mass
end

Functions can reduce global values
import "ebb"
local L = require "ebblib"
local TetLib = require 'ebb.domains.TetLib'
local dragon = TetLib.LoadTetmesh('dragon.veg')
local dt = L.Constant(L.double, 0.0002)
local K = L.Constant(L.double, 4.0)
local maxvel = L.Global(L.double, 0)
dragon.vertices:NewField('vel', L.vec3d):Load({0, 0, 0})
dragon.vertices:NewField('nxt_vel', L.vec3d):Load...
dragon.vertices:NewField('nxt_pos', L.vec3d):Load...
dragon.edges:NewField('rest_len', L.double):Load(0)
local ebb init_rest_len ( e : dragon.edges )
  var diff = e.head.pos - e.tail.pos
  e.rest_len = L.length(diff)
end
dragon.edges:foreach(init_rest_len)
-- initialize vel and acc --
local ebb compute_max_vel ( v : dragon.vertices )
  maxvel max = L.length(v.vel)
end
local ebb compute_acc ( v : dragon.vertices )
  var force = { 0.0, 0.0, 0.0 }
  -- Spring Force
  var mass = 0.0
  for e in v.edges do
    var diff = e.head.pos - v.pos
    var scale = (e.rest_len / L.length(diff)) - 1.0
    mass += e.rest_len
    force -= K * scale * diff
  end
  v.nxt_pos = v.pos + dt * v.vel + 0.5*dt*dt * force/mass
  v.nxt_vel = v.vel + dt * force/mass
end
Read Write
import "ebb"
local L = require "ebblib"

local TetLib = require 'ebb.domains.TetLib'
local dragon = TetLib.LoadTetmesh('dragon.veg')

local dt = L.Constant(L.double, 0.0002)
local K = L.Constant(L.double, 4.0)
local maxvel = L.Global(L.double, 0)

dragon.vertices:NewField('vel', L.vec3d):Load({0, 0, 0})
dragon.vertices:NewField('nxt_vel', L.vec3d):Load(...)
dragon.vertices:NewField('nxt_pos', L.vec3d):Load(...)
dragon.edges:NewField('rest_len', L.double):Load(0)

local ebb init_rest_len(e)
  var diff = e.head.pos - e.tail.pos
  e.rest_len = L.length(diff)
end
dragon.edges:foreach(init_rest_len)

-- initialize vel and acc --
local ebb compute_max_vel(v)
  maxvel = max(L.length(v.vel))
end
ebb compute_acc (v)

var force = {0.0, 0.0, 0.0}

-- Spring Force
var mass = 0.0
for e in v.edges do
  var diff = e.head.pos - v.pos
  var scale = (e.rest_len / L.length(diff)) - 1.0
  mass += e.rest_len
  force -= K * scale * diff
end

v.nxt_pos = v.pos + dt * v.vel + 0.5*dt*dt*force/mass
v.nxt_vel = v.vel + dt * force/mass

-- Spring Force
var mass = 0.0
for e in v.edges do
  var diff = e.head.pos - v.pos
  var scale = (e.rest_len / L.length(diff)) - 1.0
  mass += e.rest_len
  force -= K * scale * diff
end

v.nxt_pos = v.pos + dt * v.vel + 0.5*dt*dt*force/mass
v.nxt_vel = v.vel + dt * force/mass
import "ebb"
local L = require "ebblib"

local TetLib = require 'ebb.domains.TetLib'
local dragon = TetLib.LoadTetmesh('dragon.veg')

local dt = L.Constant(L.double, 0.0002)
local K = L.Constant(L.double, 4.0)
local maxvel = L.Global(L.double, 0)

dragon.vertices:NewField('vel', L.vec3d):Load({0, 0, 0})
dragon.vertices:NewField('nxt_vel', L.vec3d):Load({})
dragon.vertices:NewField('nxt_pos', L.vec3d):Load({})
dragon.edges:NewField('rest_len', L.double):Load(0)

local ebb init_rest_len ( e : dragon.edges )
  var diff = e.head.pos - e.tail.pos
  e.rest_len = L.length(diff)
end
dragon.edges:foreach(init_rest_len)

local ebb compute_acc ( v : dragon.vertices )
  var force = {0, 0, 0}
  -- Spring Force
  var mass = 0.0
  for e in v.edges do
    var diff = e.head.pos - v.pos
    var scale = (e.rest_len / L.length(diff)) - 1.0
    mass += e.rest_len
    force -= K * scale * diff
  end
  v.nxt_pos = v.pos + dt * v.vel
            + 0.5*dt*dt * force/mass
  v.nxt_vel = v.vel + dt * force/mass
end

-- Sim Loop
for i=1,40000 do
  dragon.vertices:foreach(compute_acc)
  dragon.vertices:swap('pos', 'nxt_pos')
  dragon.vertices:swap('vel', 'nxt_vel')
end

-- Ebb
... local particles = L.NewRelation {
    name = 'particles', size = M,
}
particles:NewField('d_cell', grid.dual_cells):Load(...) particles:NewField('pos', L.vec3f):Load(...) particles:NewField('vel', L.vec3f):Load(...) local ebb update_particle_vel ( p : particles ) var x1 = fmod( p.pos[0] - 0.5f ) var y1 = fmod( p.pos[1] - 0.5f ) var x0 = 1.0f - x1 var y0 = 1.0f - y1 p.vel = x0 * y0 * p.dual_cell.cell(0,0).vel + x1 * y0 * p.dual_cell.cell(1,0).vel + x0 * y1 * p.dual_cell.cell(0,1).vel + x1 * y1 * p.dual_cell.cell(1,1).vel end ...
for i=1,10000 do update_particle_vel(particles) update_particle_pos(particles) grid.dual_cells:point_locate(particles.dual_cell, particles.pos) end

More examples at:
http://ebblang.org

Geometric Domain Modeling

Relational Model

2d Grid
Particles
3d Grid
Triangle Mesh
Linked Chains

CPU
GPU
Cluster
import "ebb"
local L = require "ebblib"

local TetLib = require 'ebb.domains.TetLib'
local dragon = TetLib.LoadTetmesh('dragon.veg')

How was the Tetrahedral Mesh Library Implemented?

Modeling a TetMesh with Relations

Tets  Faces  Edges  Vertices
Modeling a TetMesh with Relations

A field, e.g., `pos`
Modeling a TetMesh with Relations

Topology: Connecting Elements
Topology: Connecting Elements

Example OpenGL Pipeline Input
OpenGL Pipeline Input

**Example**

```
example

OpenGL Pipeline Input

```
Indices?
Pointers?
Keys?

Indices?
Pointers?
Keys?

Indices?
Pointers?
Keys?

Indices?
Pointers?
Keys?

Triangles.vert : \texttt{GLuint}[3]

Triangles.vert : \texttt{Vert*}[3]

Triangles.vert : \texttt{Vertices}[3]

...
Key-Fields

```javascript
ebf foo(e : edges)
    var diff = e.head.pos - e.tail.pos
...
edges:NewField('head', vertices)
```
Key-Fields

```plaintext
ebb foo(e : edges)
  var diff = e.head.pos - e.tail.pos
...
```

Query-Loops

```plaintext
ebb bar(v : vertices)
  for e in v.edge(edges.tail, v) do
    v.sum_t += e.head.t
  ...
```
ebb foo( e : edges )
    var diff = e.head.pos - e.tail.pos
    ...

ebb bar( v : vertices )
    for e in L.Where(edges.tail, v) do
        v.sum_t += e.head.t
        ...
    edges:GroupBy('tail')
**Key-Fields**

```
ebb foo(e : edges)
  var diff = e.head.pos - e.tail.pos
  ...
```

**Query-Loops**

```
ebb bar(v : vertices)
  for e in v.edges do
    v.sum_t += e.head.t
  ...
```

**Affine-Indices**

```
ebb baz(c : cells)
  c.sum_p = t(Af0)np(eet[s1,0]).p + c(0,1).p \{t(0,1)}p
  + c(0,1).p \{0,1}\}, c.p +
  ...
```
Key-Fields

```
ebb foo( e : edges )
    var diff = e.head.pos - e.tail.pos
...
```

Query-Loops

```
ebb bar( v : vertices )
    for e in v.edges do
        v.sum_t += e.head.t
    ...
```

Affine-Indices

```
ebb baz( c : cells )
    c.sum_p = L.Affine(cells,
        {{1,0,1},
         {0,1,0}},
        c).p +
```

```
\[
\begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
c.x \\
c.y
\end{bmatrix}
+ 
\begin{bmatrix}
1 \\
0
\end{bmatrix}
\]
```

```
ebb baz( c : cells )
    c.sum_p = L.Affine(cells,
        {{1,0,1},
         {0,1,0}},
        c).p +
```

```
ebb bar( v : vertices )
    for e in v.edges do
        v.sum_t += e.head.t
    ...
```

```
ebb baz( c : cells )
    c.sum_p = L.Affine(cells,
        {{1,0,1},
         {0,1,0}},
        c).p +
```

```
ebb bar( v : vertices )
    for e in v.edges do
        v.sum_t += e.head.t
    ...
```

```
ebb baz( c : cells )
    c.sum_p = L.Affine(cells,
        {{1,0,1},
         {0,1,0}},
        c).p +
```

```
ebb bar( v : vertices )
    for e in v.edges do
        v.sum_t += e.head.t
    ...
```

```
ebb baz( c : cells )
    c.sum_p = L.Affine(cells,
        {{1,0,1},
         {0,1,0}},
        c).p +
```

```
ebb bar( v : vertices )
    for e in v.edges do
        v.sum_t += e.head.t
    ...
```

```
ebb baz( c : cells )
    c.sum_p = L.Affine(cells,
        {{1,0,1},
         {0,1,0}},
        c).p +
```

```
ebb bar( v : vertices )
    for e in v.edges do
        v.sum_t += e.head.t
    ...
```
ebb foo(e : edges)
var diff = e.head.pos - e.tail.pos
...

ebb bar(v : vertices)
for e in v.edges do
  v.sum_t += e.head.t
  ...

ebb baz(c : cells)
c.sum_p = L.Affine(cells, {{1,0,1}, {0,1,0}}, c).p +

Cells
c(x,y)

Dual Verts
Vertices
Dual Cells

Cells

c(x,y)

Dual Verts
Vertices
Dual Cells

Cells

c(x,y)

Dual Verts
Vertices
Dual Cells

Cells

c(x,y)
Key-Fields

```plaintext
ebb foo( e : edges )
  var diff = e.head.pos - e.tail.pos
...
```

Query-Loops

```plaintext
ebb bar( v : vertices )
  for e in v.edges do
    v.sum_t += e.head.t
  ...
```

Affine-Indices

```plaintext
ebb baz( c : cells )
  c.sum_p = c(1,0).p + c(-1,0).p
             + c(0,1).p + c(0,-1).p
...
```

1 memory read per-access
2 memory reads per-loop
0 memory reads
FEM TetMesh

Tetrahedra \( \rightarrow \) Vertices

Edges \( \rightarrow \) Vertices

head

tail

group-by: tail
Embedding in a Grid

Tetrahedra $v[4]$ → Vertices → Cells

Edges $e[4][4]$ → group-by: tail

Key-Fields

Query-Loops

Affine-Indices

$e[4][4]$
local ebb compute_acc ( v : dragon.vertices )

var force = { 0.0, 0.0, 0.0 }

-- Spring Force
var mass = 0.0
for e in v.edges do
    var diff = e.head.pos - v.pos
    var scale = (e.rest_len / L.length(diff)) - 1.0
    mass += e.rest_len
    force -= K * scale * diff
end

v.nxt_pos = v.pos + dt * v.vel
            + 0.5*dt*dt * force/mass
v.nxt_vel = v.vel + dt * force/mass
end

dragon.vertices:foreach(compute_acc)
Reductions

```plaintext
local ebb compute_max_vel ( v : vertices )
    maxvel max= L.length(v.vel)
end
```

```
for i=0,vertices.size do
    ...
end
```

trivial, given sequential access
More Implementation Details

- automated data movement and layout
- data indexing for fast-access
- code-path specialization
- subset representation

See paper & code for more details...
LULESH
Hydrodynamic Shockwave

FluidsGL
Stable Fluids

VEGA
St. Venant-Kirchhoff Elasticity

FreeFEM
Neo-Hookean Elasticity

3d Hex-Mesh

2d Grid

Particles

3d Tet-Mesh

CPU

GPU

FreeFEM

Neo-Hookean Elasticity

3d Tet-Mesh

CPU

Throughput

\textit{iterations / second}

\begin{tabular}{|l|c|}
\hline
FreeFEM & 0.024 \\
\hline
Ebb & 166.6 \\
\hline
\end{tabular}

on 2.4K tetrahedra
LULESH Hydrodynamic Shockwave

Throughput \textit{iterations / second}

Serial C Ref \hspace{1cm} 0.5

CUDA (Kepler-tuned) \hspace{1cm} 16.6

\textbf{1.25x}

all GPU code run on Kepler cards
**LULESH**  
Hydrodynamic Shockwave

![Diagram](image)

**Throughput**  
`iterations / second`

<table>
<thead>
<tr>
<th></th>
<th>Serial C Ref</th>
<th>Ebb</th>
<th>CUDA (Kepler-tuned)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.5</td>
<td>13.2</td>
<td>16.6</td>
</tr>
</tbody>
</table>

**Lines of Code**

<table>
<thead>
<tr>
<th></th>
<th>CUDA</th>
<th>Ebb</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.5K</td>
<td>1.3K</td>
</tr>
</tbody>
</table>

*all GPU code run on Kepler cards*

---

**LULESH**  
Hydrodynamic Shockwave

![Diagram](image)

**Throughput**  
`iterations / second`

<table>
<thead>
<tr>
<th></th>
<th>Serial C Ref</th>
<th>Ebb</th>
<th>CUDA (Kepler-tuned)</th>
<th>CUDA (Fermi-tuned)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.5</td>
<td>13.2</td>
<td>16.6</td>
<td>6.8</td>
</tr>
</tbody>
</table>

**Lines of Code**

<table>
<thead>
<tr>
<th></th>
<th>CUDA</th>
<th>Ebb</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.5K</td>
<td>1.3K</td>
</tr>
</tbody>
</table>

*all GPU code run on Kepler cards*
VEGA

St. Venant-Kirchhoff
Elasticity

Throughput \textit{time steps / minute}  

<table>
<thead>
<tr>
<th>Model</th>
<th>Throughput</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dragon (1.3M tets)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hose (1.3M tets)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Turtle (0.9M tets)</td>
<td></td>
<td>2.9K</td>
</tr>
</tbody>
</table>


VEGA

St. Venant-Kirchhoff
Elasticity

Throughput \textit{time steps / minute}  

<table>
<thead>
<tr>
<th>Model</th>
<th>Throughput</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dragon (1.3M tets)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hose (1.3M tets)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Turtle (0.9M tets)</td>
<td></td>
<td>2.9K</td>
</tr>
</tbody>
</table>
VEGA
St. Venant-Kirchhoff Elasticity

**Throughput** time steps / minute

<table>
<thead>
<tr>
<th>Model</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dragon (1.3M tets)</td>
<td>0.7</td>
</tr>
<tr>
<td>Hose (1.3M tets)</td>
<td>2.1</td>
</tr>
<tr>
<td>Turtle (0.9M tets)</td>
<td>2.1</td>
</tr>
</tbody>
</table>

**Lines of Code**

VEGA

- 2.9K

---

**Throughput** time steps / minute

<table>
<thead>
<tr>
<th>Model</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dragon (1.3M tets)</td>
<td>0.7</td>
</tr>
<tr>
<td>Hose (1.3M tets)</td>
<td>2.1</td>
</tr>
<tr>
<td>Turtle (0.9M tets)</td>
<td>2.1</td>
</tr>
</tbody>
</table>

**Lines of Code**

VEGA

- 2.9K multi core
- 0.8K single core
VEGA
St. Venant-Kirchhoff
Elasticity

Throughput *time steps / minute*

<table>
<thead>
<tr>
<th></th>
<th>Dragon (1.3M tets)</th>
<th>Hose (1.3M tets)</th>
<th>Turtle (0.9M tets)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GE</td>
<td>0.7</td>
<td>2.1</td>
<td>2.1</td>
</tr>
<tr>
<td>AL</td>
<td>1.6</td>
<td>3.6</td>
<td>2.1</td>
</tr>
<tr>
<td>GE</td>
<td>2.1</td>
<td>2.1</td>
<td>2.1</td>
</tr>
</tbody>
</table>

Lines of Code

<table>
<thead>
<tr>
<th></th>
<th>multi core</th>
</tr>
</thead>
<tbody>
<tr>
<td>VE</td>
<td>0.8K</td>
</tr>
<tr>
<td>NA</td>
<td>2.9K</td>
</tr>
<tr>
<td>NA</td>
<td>1.4K</td>
</tr>
</tbody>
</table>

VEGA

Throughput *time steps / minute*

<table>
<thead>
<tr>
<th></th>
<th>Dragon (1.3M tets)</th>
<th>Hose (1.3M tets)</th>
<th>Turtle (0.9M tets)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GE</td>
<td>0.7</td>
<td>2.1</td>
<td>2.1</td>
</tr>
<tr>
<td>AL</td>
<td>1.6</td>
<td>3.6</td>
<td>2.1</td>
</tr>
<tr>
<td>GE</td>
<td>2.1</td>
<td>2.1</td>
<td>2.1</td>
</tr>
</tbody>
</table>

Lines of Code

<table>
<thead>
<tr>
<th></th>
<th>multi core</th>
</tr>
</thead>
<tbody>
<tr>
<td>VE</td>
<td>0.8K</td>
</tr>
<tr>
<td>NA</td>
<td>2.9K</td>
</tr>
<tr>
<td>NA</td>
<td>1.4K</td>
</tr>
</tbody>
</table>
VEGA
St. Venant-Kirchhoff Elasticity

Throughput \( \text{time steps} / \text{minute} \)

<table>
<thead>
<tr>
<th>Model</th>
<th>Throughput</th>
<th>VEGA</th>
<th>Ebb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dragon (1.3M tets)</td>
<td>0.7</td>
<td>6.1</td>
<td>3.8 x</td>
</tr>
<tr>
<td>Hose (1.3M tets)</td>
<td>2.1</td>
<td>3.6</td>
<td>6.3 x</td>
</tr>
<tr>
<td>Turtle (0.9M tets)</td>
<td>2.1</td>
<td>2.1</td>
<td>8.7 x</td>
</tr>
</tbody>
</table>

Lines of Code

<table>
<thead>
<tr>
<th>Model</th>
<th>Multi core</th>
<th>VEGA</th>
<th>Ebb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dragon (1.3M tets)</td>
<td>0.8K</td>
<td>0.8K</td>
<td>2.9K</td>
</tr>
<tr>
<td>Hose (1.3M tets)</td>
<td>2.9K</td>
<td>2.9K</td>
<td>1.4K</td>
</tr>
<tr>
<td>Turtle (0.9M tets)</td>
<td>1.4K</td>
<td>1.4K</td>
<td>0.7K</td>
</tr>
</tbody>
</table>

http://ebblang.org

Ebb

is a programming language for writing physical simulations. Ebb programs
are performance portable: they can be efficiently executed on both CPUs
and GPUs. Ebb is embedded in the Lua programming language using Terra.
Tutorials

Introduction

These tutorials provide a tour through all the features of Ebb.

01: Hello, Ebb!

The basics of an Ebb program; print out 42 for crying out loud.

02: Domain Loading From Files

How to use a domain library to load in a mesh and render statistics and computations on that mesh; We download and use some example data.

03: Visualizing Simulations

Basic usage of VDB to generate visual output from the vertices of the octahedron.

04: User-defined Fields and Global Functions

Interoperability

These tutorials introduce the features that allow Ebb to interact with other languages, including how to write custom high-level code that interacts with Ebb.

05: Accessing Neighbors

How to access data at neighboring elements and show alternative ways of writing the code.

06: Phases, Reads, Writes, Reductions

A key feature of Ebb is that all functionality is data-driven, and these tutorials introduce the concepts of phases, reductions, and how to read, write, and reduce data.

07: Using Standard Grids

Some features of the standard grid domain, handling both periodic and normal boundary conditions.

08: Relations

Relations are the basic data structure in Ebb; these tutorials cover how to define and use relations.

09: Particle-Grid Coupling

How to connect and update the relationship between tracer particles and the underlying grid, including heat diffusion.

10: Data Layout Descriptors (DLDs)

DLDs give us the ability to access the simulation mesh and integrate a piece of unsafe code into an Ebb program.

11: Calling C-code

DLDs can also be used from C code written as unsafe C functions into an Ebb simulation.

12: File I/O

Using DLDs, we can efficiently load data into a simulation; we write code to load and write data from/to a file.

Domain Modeling

These tutorials explain the features that should be prepared to start developing models.

13: Group-By and Query-Loops

Grouping and Querying lets us invert simple to simulate heat diffusion on a graph encoded with group-by.

14: Join Tables

A common pattern that enables us to represent it to enable access to the triangles around a vertex.

15: Macros

Macros let us hide unnecessary encodings behind the join-table example using macros.

16: Grid Relations

How to use relations to represent data from a grid as a two-scale coupled grid-to-grid domain for simulations.

17: Subsets

Ebb Manual

Overview

Ebb consists of two parts: an embedded language, and a Lua API. The language proper is used to define Ebb functions, while the Lua API is used to construct and interrogate the data structures, as well as launch functions via `for` and `if` calls. For instance, in the `hello42` example program, the `print(int(a))` function is written in the Ebb language, while the rest of the program makes calls to the API.

In addition to these two parts, a set of standard domain and support libraries are provided, which this documentation will also discuss.

The remainder of the manual will assume a passing familiarity with the structure of Ebb programs. For a more intuitive introduction to the language, please see the tutorials.

Additionally this manual assumes a passing familiarity with the Lua language. Specifically, Ebb is embedded in Lua 5.1, via `Tea`. You can find a number of good tutorials, manuals and documentation online, which we will not repeat here.

The Ebb Language

The Ebb language is used to define Ebb functions, which can either be used in other Ebb functions, or executed for each element of some relation.
Why New Programming Languages for Simulation?

GILBERT BERNSTEIN
Stanford University

and

FREDRIK KJOLSTAD
Massachusetts Institute of Technology

Ebb: A DSL for Physical Simulation on CPUs and GPUs

GILBERT LOUIS BERNSTEIN and OHNAMIYEE SHAH and CRYSTAL LEMIRE and ZACHARY DEVITO and MATTHEW FISHER and PHILIP LEVIS and RTK HANRAHAN
Stanford University

This paper describes Ebb, a domain-specific language (DSL) for developing physical simulations of fluids and deformable meshes that is designed to run efficiently on both CPUs and GPUs. Ebb is implemented in the form of pre-ELLA, much as the Rosette/ 

Simit: A Language for Physical Simulation

FREDRIK KJOLSTAD
Massachusetts Institute of Technology

SHOAB KAMIL
Adobe

JONATHAN RAVAN-KELLEY
Stanford University

DAVID I. W. LEVIN
Danby Research

SHINJIRO SUEDA
California Polytechnic State University

DESAN CHEN
Massachusetts Institute of Technology

ETIENNE VOUGA
University of Texas at Austin

DANNY M. KAUFMAN
Adobe

and

GURTEJ KANWAR, WOJCIECH MATYUS, and SAMAN AMARASINGHE
Massachusetts Institute of Technology

Future Directions in Simulation Languages

Collisions, Remeshing, non-trivially parallelizable algorithms

Distributed Machines (cloud, cluster, etc.)

New Data Layout & Code Optimizations

Simulation-Specific Debugging & Support Tools
A DSL for Physical Simulation on GPUs and CPUs

Thanks to
Michael Mara,
Ivan Bermejo-Moreno,
& Thomas D. Economon

http://ebblang.org

Gilbert Bernstein
Chinmayee Shah
Crystal Lemire
Zach DeVito
Matthew Fisher
Philip Levis
Pat Hanrahan