Jen-Hsun’s Argument for Co-Processing

Power efficiency
- CPUs more efficient for sequential workloads
- GPUs more efficient for data-parallel workloads

Amdahl’s Law
- Applications have a mixture of sequential and parallel code
- Parallelism may be limited by sequential performance

Therefore,
- Optimal platform involve both fast sequential cores plus fast parallel cores
ZuneHD uses Tegra

Tegra APX2600: CPU+GPU+HD+...

CPU: ARM11 MPCore
GPU: ULP
HD: 720p decoder

...
Platforms even more Heterogeneous

Cluster (e.g. 100K node cluster)
- Distributed memory
- MPI for communication

Multi-core SMP (e.g. 32 core, 4-socket systems)
- Shared memory, transaction memory
- Threads/locks, OpenMP

Many-core GPU (e.g. Fermi)
- Separate GPU memory
- SIMT programming model (CUDA)

+ combinations of these systems (accelerators)

Is it Possible to Write One Program and Run it on all these Machines?
HYPOTHESIS: YES, BUT NEED

Domain Specific
Programming Environments
and
Languages

Domain-Specific Languages

Widely used in many application areas

- matlab / R
- SQL / Microsoft’s LINQ / map-reduce
- OpenGL/D3D and Cg/HLSL
- ...

DSLs are a hot topic now

- Programming language community (C#, Scala)
- Web programming environments (Ruby)
Three Advantages

Productivity
- Separate domain expertise (computational science) from computer science expertise

Portability
- Run on wide range of platforms

Performance
- Super-optimize using a combination of domain knowledge and platform knowledge

RenderMan Shading Language

surface corrode(float Ks=0.4, Ka=0.1, rough=0.25)
{
    float i, freq=1, turb=0;
    // compute fractal texture
    for( i=0; i<6; i++ ) {
        turb+=1/freq*noise(freq*P);
        freq*=2;
    }
    // perturb surface
    P -= turb * normalize(N);
    N = faceforward(normalize(calculatenormal(P)));
    // compute reflection and final color
    Ci = Cs*(Ka*ambient()+Ks*specular(N,I,rough));
}
Graphics Libraries

```c
glMatrixMode(GL_PROJECTION);
glPerspective(45.0);
for( ;; ) {
    glBegin(TRIANGLES);
    glVertex(...);
    glVertex(...);
    ...
    glEnd();
}
glSwapBuffers();
```

OpenGL “Grammar”

```plaintext
<Scene> = <BeginFrame> <Camera> <World> <EndFrame>

<Camera> = glMatrixMode(GL_PROJECTION) <View>
<View> = glPerspective | glOrtho

<World> = <Objects>*
<Object> = <Transforms>* <Geometry>
<Transforms> = glTranslatef | glRotatef | ...
<Geometry> = glBegin <Vertices> glEnd
<Vertices> = [glColor] [glNormal] glVertex
```
Productive, Performant, and Portable

Easy to write OpenGL programs (CG101)

Runs at extremely high speed
- Graphics drivers do amazing things
- Graphics chips do amazing things

Runs across very different architectures
- Wide range of GPUs on the market
- Backward and forward compatible

Three Four Advantages

Productivity
Portability
Performance
Encourage innovation
  - Allow NVIDIA to radically optimize implementation
  - Reduce cost of porting if the low-level programming model changes
DSLs in Other Parallel Applications?

Future graphics pipelines with rendering and physics
Statistics/machine learning and data analysis (beyond R)
Physical simulation
Computer vision and imaging
Brain simulation
Autonomous vehicles
...

Liszt

Z. Devito, M. Medina, M. Barrientos, J. Alonso, E. Darve, F. Ham
P. Hanrahan

“...the most technically advanced and perhaps greatest pianist of all time...” 1811-1886
Characterize the operability limits of a hypersonic propulsion system using predictive computations. Primary focus is the unstart phenomena triggered by thermal choking in a hydrogen-fueled scramjet.

Stanford DOE PSAAP Center

State-of-the-art unstructured RANS solver
– Main tool for system-level simulation

Comparison of Joe simulation result to experiment: DLR ground test based on atmospheric conditions at 27 km
Typical Joe C Code Fragment

```c
for( int ifa = zone->ifa_f; ifa <= zone->ifa-1; ifa++ ) {
    int naf_f = noofa_i[ifa];
    intnof_l = noofa_i[ifa+1] - 1;
    int nnof = nof_l - naf_f + 1;
    double x_fa[3], u_fa_bc[3];
    for( int i = 0; i< 3; i++ ) {
        x_fa[i] = 0.0;
        u_fa_bc[i] = 0.0;
    }
    double phi_fa_bc = 0.0;
    for( int nof = naf_f; nof <= nof_l; nof++ ) {
        int ino = noofa_v[nof];
        int ibc = bc_flag[ino];
        no_flag[ino] = nof - naf_f;
        for( int i = 0; i< 3; i++ ) {
            x_fa[i] += x_no[ino][i];
            u_fa_bc[i] += ubc->data[ibc][i];
        }
        phi_fa_bc += phibc->data[ibc];
    }
}
```

Joe in Liszt (Scala Version)

```scala
val pos = new Field[Vertex,double3]
val A = new SparseMatrix[Vertex,Vertex]

for (c <- mesh.cells) {
    val center = avg pos of c.vertices
    for (f <- c faces) {
        val face_dx = avg pos of f.vertices - center
        for (e <- f edgesWith c CounterClockwise) {
            val v0 = e.tail
            val v1 = e.head
            val v0_dx = pos(v0) - center
            val v1_dx = pos(v1) - center
            val face_normal = v0_dx cross v1_dx
            // calculate flux for face ...
            A(v0,v1) += ...
            A(v1,v0) -= ...
        }
    }
}
```
Built-in Features

Objects
- Short vectors, vectors, dense and sparse matrices
- Mesh, cells, faces, edges, vertices
- Fields
- Sets (unordered) and lists (ordered)

Solvers
- Sparse matrix solvers (e.g. hypre, PETSc, …)

Parallel iteration
- Map, reduce, …
Domain-Specific Optimizations

1. Automatic partitioning
2. Automate maintenance of distributed ghost cells
3. Optimize mesh representation
4. Optimize storage of fields
5. …

Liszt uses program analysis to perform these optimizations
Program Analysis of Neighborhoods

\[
\text{Field<Cell, double> rho;}
\]
\[
\text{Field<Face, double> rhoOutside;}
\]

forall (Face f in mesh.faces()) {
    rhoOutside(f) = calc_flux(f, rho(f.outside()))
                   + calc_flux(f, rho(f.inside()));
}

Program Analysis of Neighborhoods

Node 0
Owned Cells
Ghost Cells
Program Analysis of Neighborhoods

Field<Cell, double> rho;
Field<Face, double> rhoOutside;

forall (Face f in mesh.faces()) {
    Cell outside = f.outside();
    rhoOutside(f) = 0.5 * rho(c));
    forall (Cell c in outside.cells()) {
        rhoOutside(f) += .25 * rho(c);
    }  
}

Program Analysis of Neighborhoods

Field<Cell, double> rho;
Field<Face, double> rhoOutside;

forall (Face f in mesh.faces()) {
    Cell outside = f.outside();
    rhoOutside(f) = 0.5 * rho(c));
    forall (Cell c in outside.cells()) {
        rhoOutside(f) += .25 * rho(c);
    }  
}
Domain Decomposition / Ghost Cells

Liszt creates a graph of mesh adjacencies needed to run the algorithm.
Graph is handed to Parmetis to determine optimal partition.
Communication of information in ghost cells is also automatically handed.

Choose Mesh Representation

1. Optimize based on what topological relationships are used by the simulation code.
2. Optimize based on special structure of the input mesh.

Input Meshes

- Unstructured
- Triangular Faces
- Regular Grid
Choose Mesh Representation (1)

// Finite volume method
forall(Cell c in mesh.cells()) {
    // calculate gradients
}
forall(Face f in mesh.faces()) {
    double flux = //flux for face
    data(f.inside()) -= flux;
    data(f.outside()) += flux;
}
forall(Cell c in mesh.cells()) {
    // update cell state
}
Choose Mesh Representation (2)

forall( Face f in mesh.faces() ) {
    forall( Face f2 in f.faces() ) {
        field(f)++;
    }
}

Input Meshes

Mesh 1 – Unstructured

forall( Face f in mesh.faces() ) {
    forall( Face f2 in f.faces() ) {
        field(f)++;
    }
}
Mesh 1 – Unstructured

forall(Face f in mesh.faces()) {
    forall(Face f2 in f.faces()) {
        field(f)++;
    }
}

Winged Edge Data Structure

struct Face {
    Edge *e;
    int id;
};

struct Edge {
    Vert *v0, *v1;
    Face *f0, *f1;
    Edge *e0, *e1,
        *e2, *e3;
};

for(int i = 0; i < nfaces; i++) {
    Face * f = faces[i];
    Edge * e = f->e;
    do {
        Face * f2 = faceAroundEdge(e, f);
        field[f2->id]++;
    } while( (e = edgeCCWAroundFace(e, f))
        != f->e );
}

Mesh 1 – Unstructured

forall(Face f in mesh.faces()) {
    forall(Face f2 in f.faces()) {
        field(f)++;
    }
}

Winged Edge Data Structure

struct Face {
    Edge *e;
    int id;
};

struct Edge {
    Vert *v0, *v1;
    Face *f0, *f1;
    Edge *e0, *e1,
        *e2, *e3;
};

for(int i = 0; i < nfaces; i++) {
    Face * f = faces[i];
    Edge * e = f->e;
    do {
        Face * f2 = faceAroundEdge(e, f);
        field[f2->id]++;
    } while( (e = edgeCCWAroundFace(e, f))
        != f->e );
}
Mesh 2 – Triangular Faces

forall(Face f in mesh.faces() ) {
    forall(Face f2 in f.faces() ) {
        field(f)++;
    }
}

Triangle mesh representation

struct Face {
    Edge *e0,*e1,*e2;
    Face *f0,*f1,*f2;
    Vert *v0,*v1,*v2;
    int id;
};
Mesh 2 – Triangular Faces

```
for(int i = 0; i < nfaces; i++) {
    Face * f = faces[i];
    field[f->f0->id]++;
    field[f->f1->id]++;
    field[f->f2->id]++;
}
```

Triangle mesh representation

```c
struct Face {
    Edge *e0,*e1,*e2;
    Face *f0,*f1,*f2;
    Vert *v0,*v1,*v2;
    int id;
};
```

Mesh 3 – Regular Grid

```
forall(Face f in mesh.faces() ) {
    forall(Face f2 in f.faces() ) {
        field(f)++;
    }
}
```
Mesh 3 – Regular Grid

forall(Face f in mesh.faces() ) {
    forall(Face f2 in f.faces() ) {
        field(f)++;
    }
}

Simple mesh data structure

int nfaces = faces_x * faces_y;
int field[faces_x][faces_y];

Mesh 3 – Regular Grid

for(int x = 1; x < faces_x - 1; x++) {
    for(int y = 1; y < faces_y - 1; y++) {
        field[x+1][y]++;
        field[x-1][y]++;
        field[x][y+1]++;
        field[x][y-1]++;
    }
}

• Can vectorize the loop body to handle 4 faces at once
• Can do intelligent blocking of the matrix operations

Simple mesh data structure

int nfaces = faces_x * faces_y;
int field[faces_x][faces_y];
### Array of Structs vs. Struct of Arrays

<table>
<thead>
<tr>
<th>K1</th>
<th>K2</th>
<th>K3</th>
<th>K4</th>
<th>K5</th>
<th>K6</th>
<th>K7</th>
</tr>
</thead>
<tbody>
<tr>
<td>enthalpy</td>
<td>gamma</td>
<td>pressure</td>
<td>rhoE</td>
<td>rhou</td>
<td>rho</td>
<td>RoM temperature</td>
</tr>
<tr>
<td></td>
<td>muLam temperature</td>
<td>Lambda-OverCp</td>
<td>muLam</td>
<td>local_dt</td>
<td>rho</td>
<td>rhou</td>
</tr>
<tr>
<td></td>
<td>gamma</td>
<td>pressure</td>
<td>rho</td>
<td>local_dt</td>
<td>rhou</td>
<td>rho</td>
</tr>
<tr>
<td></td>
<td>muLam</td>
<td>pressure</td>
<td>rho</td>
<td>local_dt</td>
<td>rhou</td>
<td>rho</td>
</tr>
</tbody>
</table>

- Fields accessed per kernel
- Prob[B access|A access]
- Determine optimal field layouts

### Target Application Areas

Future rendering pipelines (Gramps)
Data-parallel programming (Kore)
Statistics/machine learning - beyond R (OptiML)
Physical simulation (Liszt)
...
Computer vision and imaging
Brain simulation
Autonomous vehicles
...
How to Create Domain-Specific Languages

“Little” Languages (UNIX)

e.g. sed, awk, sh, Renderman, matlab, R, ...

Typically built using compiler generators: yacc and lex

Disadvantages:
- Continual requests for more generality
- Proliferation of syntax: “$x”
- Cannot use multiple DSLs in the same application
Domain-Specific Embedded Language

Examples: Lisp, C++ templates, C#, Haskell, Ruby, Scala

Advantages:
+ Consistent base syntax
+ Multiple DSLs may interoperate

Rake – Make in Ruby

SRC = FileList['*.c']
OBJ = SRC.ext('o')

task :default => ['hello']
rule '.o' => '.c' do |t|
  sh "cc -c -o #{t.name} #{t.source}"
end
file "hello" => OBJ do
  sh "cc -o hello #{OBJ}"
end

# File dependencies go here ...
file 'main.o' => ['main.c', 'greet.h']
file 'greet.o' => ['greet.c']
Rake – Make in Ruby

Example Rakefile

Features of Ruby for DSELs
- Overloaded operators
- Simple (Smalltalk-like) syntax: a b c -> a.b(c)
- Blocks: Lambdas and closures
- Dynamic typing

LINQ (Language-Integrated Query)

```csharp
var personsNotInSeattle =
    from p in person
    where p.Address != "Seattle"
    orderby p.FirstName
    select p;
```
## LINQ Driving Feature Set for C# 3.0

**Language extensions**
- Implicitly typed variables
- Lambdas
- Anonymous classes
- Extensions (implicit type wrapper)

Enables parallel data analysis
- SQL engines
- PLINQ: SMP
- DryadLINQ: Clusters

## Domain-Specific Embedded Language

### Approach 1: Objects w/ operators
+ Access to most embedding language features
+ Compiled code
- No general system for program analysis or program transformations
Domain-Specific Embedded Language

Approach 2: Build expression tree and interpret
+ Allows program analysis and transformation
+ Extend semantics of embedding language
- Hides some features of the embedding language (e.g. types and type checking)
- Slow because it is interpreted

Integrated DSEL Environment

1. Flexible syntax and grammar
   - The flexibility of lisp macros in a modern language
   - Convert to simple abstract syntax tree
2. Program analysis
   - Library supporting domain-specific analysis
3. Program transformation
   - Domain specific rewrite rules
4. Code generation
   - Generate code for different parallel platforms
5. Language workbench / IDE
### DELITE Approach to DSELs

- Support DSELs by embedding in Scala
- Build dataflow graph by deferring method execution
- Use static analysis to complement dynamic optimizations
- Apply a variety of generic and domain transformation to dataflow graph
- Intelligently schedule work to maximize locality with domain knowledge

### DSELs in Scala

Scala combines both the strengths of FP and OOP
- Encourages use of best approach for a particular problem

Designed to embed DSLs
- Concise syntax, implicit type conversions, ...

**Functional programming**
- Higher-order functions allow parallel control structures (fork,...)
- Declarative programming style ⇒ exposes parallelism

**Object-oriented programming**
- Familiar programming model
- Allows mutable data structures

Scala compiles to Java bytecode
- JVM platform is mature and high performance
- Allows the use of Java libraries
PPL Overview

Applications
- Scientific Engineering
- Virtual Worlds
- Personal Robotics
- Data informatics

Domain Specific Languages
- Rendering
- Physics
- Scripting
- Probabilistic Reasoning
- Machine Learning

Common Parallel Runtime
- Explicit / Static
- Implicit / Dynamic

Parallel Object Language

Heterogeneous Hardware
- OOO Cores
- SIMD Cores
- Threaded Cores
- Programmable Hierarchies
- Scalable Coherence
- Isolation & Atomicity
- Pervasive Monitoring

Acknowledgements

Bill Dally
Alex Aiken
Kunle Olukotun
Eric Darve
Juan Alonso
Frank Ham
Vijay Pande
Arthur Whitney
Zach Devito
Montse Medina
Michael Barrientos
Nathan Bronson
Hassan Chafi
Thank you