Gift of Gab
Dreamin

“Someday... the perfect auto-parallelizing compiler will exist.”

- Timothy Parker
Course themes:

Designing computer systems that **scale**
(running faster given more resources)

Designing computer systems that are **efficient**
(running faster under constraints on resources)

Techniques discussed:
- Exploiting parallelism in applications
- Exploiting locality in applications
- Leveraging hardware specialization (last time)
Hardware trend: specialization of execution

- **Multiple forms of parallelism**
  - SIMD/vector processing
  - Multi-threading
  - Multi-core
  - Multiple node
  - Multiple server
  
  Fine-granularity parallelism: perform same logic on different data
  Mitigate inefficiencies (stalls) caused by unpredictable data access
  Varying scales of coarse-granularity parallelism

- **Heterogeneous execution capability**
  - Programmable, latency-centric (e.g., “CPU-like” cores)
  - Programmable, throughput-optimized (e.g., “GPU-like” cores)
  - Fixed-function, application-specific (e.g., image/video/audio processing)

Motivation for parallelism and specialization: maximize compute capability given constraints on chip area, chip energy consumption.
Result: amazingly high compute capability in a wide range of devices!
Claim: most software uses modern hardware resources inefficiently

- Consider a piece of sequential C code
  - Let’s consider the performance of this code “baseline performance”

- Well-written sequential C code: ~ 5-10x faster
- Assembly language program: another small constant factor faster
- Java, Python, PHP, etc. ??

Credit: Pat Hanrahan
Code performance: relative to C (single core)

GCC -O3 (no manual vector optimizations)

Data from: The Computer Language Benchmarks Game: http://shootout.alioth.debian.org
Even good C code is inefficient

Recall Assignment 1’s Mandelbrot program

Consider execution on this laptop: quad-core, Intel Core i7, AVX instructions...

Single core, with AVX vector instructions: 5.8x speedup over C implementation

Multi-core + hyper-threading + AVX instructions: 21.7x speedup

Conclusion: basic C implementation compiled with -O3 leaves a lot of performance on the table
Making efficient use of modern machines is challenging

(proof by assignments 2, 3, and 4)

In our assignments you only programmed homogeneous parallel computers. And parallelism in that context was not easy.

Assignment 2: GPU cores only

Assignment 3: Blacklight (multiple CPUs with relatively fast interconnect)

Assignment 4: multiple parallel machines
Recall from last time: need for efficiency leading to heterogeneous parallel platforms

Integrated CPU + GPU

**GPU:**
throughput cores + fixed-function

**Qualcomm Snapdragon SoC 800 Processor**
- Ultra HD Capture and Playback
- D7SHD and Dolby
- Digital Plus audio
- Expanded Gestures
- 35MP with dual ISP
- Support for up to 2560x2048 display
- Miracast 1080p
- HD support
- LTE GNSS with support for three GPS constellations

Mobile system-on-a-chip:
CPU+GPU+media processing
Hardware diversity is a huge challenge

- Machines with very different performance characteristics
- Even worse: different technologies and performance characteristics within the same machine at different scales
  - Within a core: SIMD, multi-threading: fine-granularity sync and communication
  - Across cores: coherent shared memory via fast on-chip network
  - Hybrid CPU+GPU multi-core: incoherent (potentially) shared memory
  - Across racks: distributed memory, multi-stage network
Variety of programming models to abstract HW

- Machines with very different performance characteristics
- Worse: different technologies and performance characteristics within the same machine at different scales
  - Within a core: SIMD, multi-threading: fine grained sync and comm.
    - Abstractions: SPMD programming (ISPC, Cuda, OpenCL)
  - Across cores: coherent shared memory via fast on-chip network
    - Abstractions: OpenMP shared address space, Cilk, TBB
  - Hybrid CPU+GPU multi-core: incoherent (potentially) shared memory
    - Abstractions: OpenCL
  - Across racks: distributed memory, multi-stage network
    - Abstractions: message passing (MPI, Go channels, Charm++)

Credit: Pat Hanrahan
Huge challenge

- Machines with very different performance characteristics
- Worse: different performance characteristics within the same machine at different scales
- To be efficient, software must be optimized for HW characteristics
  - Difficult even in the case of one level of one machine **
  - Combinatorial complexity of optimizations when considering a complex machine, or different machines
  - Loss of software portability

** Little success developing automatic tools to identify efficient HW mappings for arbitrary, complex applications

Credit: Pat Hanrahan
Open computer science question:

How do we enable programmers to write software that efficiently uses current and future heterogeneous, parallel machines?
The [magical] ideal parallel programming language

- High Performance (software is scalable and efficient)
- Completeness (applicable to most problems we want to write a program for)
- Productivity (ease of development)

Credit: Pat Hanrahan
Successful programming languages

Here: definition of success = widely used

- High Performance (software is scalable and efficient)
- Productivity (ease of development)
- Completeness (applicable to most problems we want to write a program for)

Credit: Pat Hanrahan
Growing interest in domain-specific programming systems

To realize high performance and productivity: willing to sacrifice completeness

Credit: Pat Hanrahan
Domain-specific programming systems

- Main idea: raise level of abstraction for expressing programs

- Introduce high-level programming primitives specific to an application domain
  - Productive: intuitive to use, portable across machines, primitives correspond to behaviors frequently used to solve problems in targeted domain
  - Performant: system uses domain knowledge to provide efficient, optimized implementation(s)
    - Given a machine: system knows what algorithms to use, parallelization strategies to employ for this domain
    - Optimization goes beyond efficient mapping of software to hardware! The hardware platform itself can be optimized to the abstractions as well

- Cost: loss of generality/completeness
Two domain-specific programming examples

1. Liszt: scientific computing
2. Halide: image processing

(Bonus slides contain a third example: OpenGL)

(Also, SQL is another good example)
Example 1:
Lizst: a language for solving PDE’s on meshes

[DeVito et al. Supercomputing 11, SciDac ’11]

Slide credit for this section of lecture:
Pat Hanrahan and Zach Devito (Stanford)

http://liszt.stanford.edu/
Fields on unstructured meshes

val Position = FieldWithLabel[Vertex, Float3]("position")

val Temperature = FieldWithConst[Vertex, Float](0.0f)
val Flux = FieldWithConst[Vertex, Float](0.0f)
val JacobiStep = FieldWithConst[Vertex, Float](0.0f)

Notes:
Fields are a higher-kindied type
(special function that maps a type to a new type)
Explicit algorithm: heat conduction on grid

```scala
var i = 0;
while ( i < 1000 ) {
    Flux(vertices(mesh)) = 0.f;
    JacobiStep(vertices(mesh)) = 0.f;
    for ( e <- edges(mesh) ) {
        val v1 = head(e)
        val v2 = tail(e)
        val dP = Position(v1) - Position(v2)
        val dT = Temperature(v1) - Temperature(v2)
        val step = 1.0f/(length(dP))
        Flux(v1) += dT*step
        Flux(v2) -= dT*step
        JacobiStep(v1) += step
        JacobiStep(v2) += step
    }
    i += 1
}
```

Coloring key:
- Fields
- Mesh
- Topology functions
- Iteration over set
Liszt’s topological operators

**BoundarySet**
- `BoundarySet`[ME <: MeshElement](name : String) : Set[ME]

**Vertices**
- `vertices(e : Mesh) : Set[Vertex]`

**Cells**
- `cells(e : Mesh) : Set[Cell]`

**Edges**
- `edges(e : Mesh) : Set[Edge]`

**Faces**
- `faces(e : Mesh) : Set[Face]`

**Vertices**
- `vertices(e : Vertex) : Set[Vertex]`

**Cells**
- `cells(e : Vertex) : Set[Cell]`

**Edges**
- `edges(e : Vertex) : Set[Edge]`

**Faces**
- `faces(e : Vertex) : Set[Face]`

**Vertices**
- `vertices(e : Edge) : Set[Vertex]`

**Cells**
- `cells(e : Edge) : Set[Cell]`

**Heads**
- `head(e : Edge) : Vertex`

**Tails**
- `tail(e : Edge) : Vertex`

**Flips**
- `flip(e : Edge) : Edge`

**Towardss**
- `towards(e : Edge, t : Vertex) : Edge`

**Cells**
- `cells(e : Cell) : Set[Cell]`

**Vertices**
- `vertices(e : Cell) : Set[Vertex]`

**Faces**
- `faces(e : Cell) : Set[Face]`

**Edges**
- `edges(e : Cell) : Set[Edge]`

**Inside**
- `inside(e : Face) : Cell`

**Outside**
- `outside(e : Face) : Cell`

**Flips**
- `flip(e : Face) : Face`

**Towardss**
- `towards(e : Face, t : Cell) : Face`
Liszt programming

- A Liszt program describes operations on fields of an abstract mesh representation
- Application specifies type of mesh (regular, irregular) and its topology
- Mesh representation is chosen by Liszt (not by the programmer)
  - Based on mesh type, program behavior, and machine
Compiling to parallel computers

Recall challenges you have faced in your assignments

1. Identify parallelism
2. Identify data locality
3. Reason about required synchronization
Key: determining program dependencies

1. Identify parallelism
   - Absence of dependencies implies code can be executed in parallel

2. Identify data locality
   - Partition data based on dependencies (localize dependent computations for faster synchronization)

3. Reason about required synchronization
   - Synchronization is needed to respect existing dependencies (must wait until the values a computation depends on are known)

In general programs, compilers are unable to infer dependencies at global scale: \( a[f(i)] += b[i] \) (must execute \( f(i) \) to know if dependency exists across loop iterations \( i \) )
Liszt is constrained to allow dependency analysis

Inferring “stencils”: “stencil” = mesh elements accessed in an iteration of loop = dependencies for the iteration

Statically analyze code to find stencil of each top-level for loop

- Extract nested mesh element reads
- Extract field operations

```scala
for (e <- edges(mesh)) {
  val v1 = head(e)
  val v2 = tail(e)
  val dP = Position(v1) - Position(v2)
  val dT = Temperature(v1) - Temperature(v2)
  val step = 1.0f/(length(dP))
  Flux(v1) += dT*step
  Flux(v2) -= dT*step
  JacobiStep(v1) += step
  JacobiStep(v2) += step
}
```

Edge 6’s read stencil is F and D
Restrict language for dependency analysis

Language Restrictions:

- Mesh elements are only accessed through built-in topological functions:
  
  `cells(mesh), ...`

- Single static assignment:

  ```
  val v1 = head(e)
  ```

- Data in fields can only be accessed using mesh elements:

  ```
  Pressure(v)
  ```

- No recursive functions

Allows compiler to automatically infer stencil for a loop iteration.
Portable parallelism: use dependencies to implement different parallel execution strategies

Strategy 1: mesh partitioning
- Assign partition to each computational unit
- Use *ghost* elements to coordinate cross-boundary communication.

Strategy 2: Mesh coloring
- Calculate interference between work items on domain
- Schedule work-items into non-interfering batches
Consider distributed memory implementation
Store region of mesh on each node in a cluster
(Note: ParMETIS is a tool for partitioning meshes)
Each processor also needs data for neighboring cells to perform computation (“ghost cells”).
Listz allocates ghost region storage and emits required communication.
GPU implementation: parallel reductions

Previous example, one region of mesh per processor (or node in MPI cluster)
On GPU, natural parallelization is one edge per CUDA thread

Threads

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory

for (e <- edges(mesh)) {
  ...
  Flux(v1) += dT*step
  Flux(v2) -= dT*step
  ...
}

Different edges share a vertex: requires atomic update of per-vertex field data
GPU implementation: conflict graph

Threads 1 edge assigned to 1 thread

Identify mesh edges with colliding writes (lines in graph indicate presence of collision)

Can simply run program once to get this information. (results valid for subsequent executions provided mesh does not change)
GPU implementation: conflict graph

Threads 1 edge assigned to 1 thread

Memory

“Color” nodes in graph such that no connected nodes have the same color

Can execute on GPU in parallel, without atomic operations, by running all nodes with the same color in a single CUDA launch.
MPI performance of Lizst program
256 nodes, 8 cores per node

Important: performance portability!
Same Liszt program also runs with high efficiency on GPU (results not shown here).
But uses a different algorithm when compiled to GPU! (graph coloring)
Liszt summary

- **Productivity:**
  - Abstract representation of mesh: vertices, edges, faces, fields (concepts that a scientist thinks about already!)
  - Intuitive topological operators

- **Portability**
  - Same code runs on cluster of CPUs (MPI runtime) and GPUs

- **High-Performance**
  - Language is constrained to allow compiler to track dependencies
  - Used for locality-aware partitioning in distributed memory implementation
  - Used for graph coloring in GPU implementation
  - Completely different parallelization strategies for different platforms
  - Underlying mesh representation can be customized by system based on usage and platform (e.g., struct of arrays vs. array of structs)
Example 2:

Halide: a domain-specific language for image processing

Slide acknowledgments:
Jonathan Ragan-Kelley (MIT)
What does this C++ code do?

Total work $\sim 6 \times \text{width()} \times \text{height()}

```cpp
void blur(const Image &in, Image &blurred) {
    Image tmp(in.width(), in.height());

    for (int y = 0; y < in.height(); y++)
        for (int x = 0; x < in.width(); x++)
            tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

    for (int y = 0; y < in.height(); y++)
        for (int x = 0; x < in.width(); x++)
            blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
}
```

3x3 convolution (work efficient, two-pass implementation)
$\sim 9.9$ microseconds per pixel on a modern CPU
3x3 box blur

2X zoom view
Optimized C++ code: 3x3 image blur

Good: 10x faster: ~ 0.9 microsec per pixel on a modern quad-core CPU

Bad: specific to SSE, hard to tell what's going on at all!

```cpp
void fast_blur(const Image &in, Image &blurred) {
    _m128i one_third = _mm_set1_epi16(21846);
    #pragma omp parallel for
    for (int yTile = 0; yTile < in.height(); yTile += 32) {
        _m128i a, b, c, sum, avg;
        _m128i tmp[(256/8)*(32+2)];
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            _m128i *tmpPtr = tmp;
            for (int y = -1; y < 32+1; y++) {
                const uint16_t *inPtr = &in(xTile, yTile+y);
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_loadu_si128((__m128i*)(&inPtr[1]));
                    b = _mm_loadu_si128((__m128i*)(&inPtr[0]));
                    c = _mm_loadu_si128((__m128i*)(&inPtr[2]));
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(tmpPtr++, avg);
                    inPtr += 8;
                }
            }
            tmpPtr = tmp;
            for (int y = 0; y < 32; y++) {
                _m128i *outPtr = (_m128i*)(&blurred(xTile, yTile+y));
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_load_si128(tmpPtr+(2*256)/8);
                    b = _mm_load_si128(tmpPtr+256/8);
                    c = _mm_load_si128(tmpPtr++);
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(outPtr++, avg);
                }
            }
        }
    }
}
```
Halide blur

Halide = two domain-specific co-languages

1. A purely functional DSL for defining image processing algorithms
2. A DSL for defining “schedules” for how to map these algorithms to a machine

Images are pure functions from integer coordinates (up to 4D domain) to values (color of corresponding pixels)

Algorithms are a series of functions (think: pipeline stages)
Functions (side-effect-free) map coordinates to values (in, tmp and blurred are functions)

```
Func halide_blur(Func in) {
    Func tmp, blurred;
    Var x, y, xi, yi;

    // The algorithm
    tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
    blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;

    return blurred;
}
```

NOTE: execution order and storage are unspecified by the abstraction. The implementation can evaluate, reevaluate, cache individual points as desired!
Consider a Halide program as a pipeline

```c
Func halide_blur(Func in) {
    Func tmp, blurred;
    Var x, y, xi, yi;

    // The algorithm
    tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
    blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;

    return blurred;
}
```
Halide blur

- Halide = two domain-specific co-languages
  1. A purely functional DSL for defining image processing algorithms
  2. A DSL for defining “schedules” for how to map these algorithms to a machine

```c
Func halide_blur(Func in) {
    Func tmp, blurred;
    Var x, y, xi, yi;

    // The algorithm
    tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
    blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;

    // The schedule
    blurred.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
    tmp.chunk(x).vectorize(x, 8);
    return blurred;
}
```

When evaluating `blurred`, use 2D tiling order (loops named by `x`, `y`, `xi`, `yi`). Use tile size `256 x 32`.

Vectorize the `xi` loop (8-wide), use threads to parallelize the `y` loop.

Produce only chunks of `tmp` at a time.

Vectorize the `x` (innermost) loop.
Separation of algorithm from schedule

- **Key idea:** separate specification of image processing algorithm (machine independent) from specification of schedule (machine-dependent mapping)

- **Given algorithm and schedule description,** Halide generates very high-quality code for a target machine
  - **Domain scope:**
    - All computation must be over regular (up to 4D) grids
    - Only feed-forward pipelines (includes special support for reductions and fixed recursion depth)
    - All dependencies are inferable by compiler
Halide schedule: producer/consumer scheduling

- Four basic scheduling primitives shown below
- Fifth primitive: “reuse” not shown

**breadth first:** each function is entirely evaluated before the next one.

**sliding window:** values are computed when needed then stored until not useful anymore.

**total fusion:** values are computed on the fly each time that they are needed.

**tiles:** overlapping regions are processed in parallel, functions are evaluated one after another.
## Halide schedule: domain iteration

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td></td>
</tr>
</tbody>
</table>

**serial y, serial x**

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>13</td>
<td>19</td>
<td>25</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>14</td>
<td>20</td>
<td>26</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>15</td>
<td>21</td>
<td>27</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>16</td>
<td>22</td>
<td>28</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>17</td>
<td>23</td>
<td>29</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>18</td>
<td>24</td>
<td>30</td>
<td>36</td>
<td></td>
</tr>
</tbody>
</table>

**serial x, serial y**

Specify both order and how to parallelize (multi-thread, SIMD vector)

2D blocked iteration order

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

**serial y, vectorized x**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

**parallel y, vectorized x**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

**split x into 2x_o + x_i, split y into 2y_o + y_i, serial y_o, x_o, y_i, x_i**
Halide results

- Camera RAW processing pipeline
  (Convert RAW sensor data to RGB image)
  - Original: 463 lines of hand-tuned ARM assembly
  - Halide: 2.75x less code, 5% faster

- Bilateral filter
  (Common image filtering operation used in many applications)
  - Original 122 lines of C++
  - Halide: 34 lines algorithm + 6 lines schedule
    - CPU implementation: 5.9x faster
    - GPU implementation: 2x faster than hand-written CUDA

Takeaway: Halide is not magic, but its abstractions allow rapid manual exploration of optimization space, allowing programmer to reach optimal points quickly
Many other recent domain-specific programming systems

Less domain specific than examples given today, but still designed specifically for:
- data-parallel computations on big data for distributed systems (“Map-Reduce”)
- Model-view-controller paradigm for web-applications

Also see **Green-Marl**
(another DSL for describing operations on graphs)

**Hadoop**
DSL for graph-based machine learning computations

Ongoing efforts in many domains...
Domain-specific programming system development

- Can develop DSL as a stand-alone language
  - Graphics shading languages
  - MATLAB, SQL

- “Embed” DSL in an existing generic language
  - e.g., C++ library (GraphLab, OpenGL host-side API, Map-Reduce)
  - Lizst syntax above was all valid Scala code

- Active research idea:
  - Design generic languages that have facilities that assist rapid embedding of new domain-specific languages
Facilitating development of new domain-specific languages

“Embed” domain-specific language in generic, flexible embedding language (Stand-alone domain-special language must implement everything from scratch)

Typical Compiler

“Modular staging” approach:

Domain language adopts front end from highly expressive embedding language

But customizes intermediate representation (IR) and participates in backend optimization and code-generation phases (exploiting domain knowledge while doing so)

Leverage techniques like operator overloading, modern OOP (traits), type inference, closures, to make embedding language syntax appear native:

Liszt code shown before was actually valid Scala!

Credit: Hassan Chafi
Summary

- **Modern machines: parallel and heterogeneous**
  - Only way to increase compute capability in energy-constrained world

- **Most software uses very little of peak capability of machine**
  - Very challenging to tune programs to these machines
  - Tuning efforts are not portable across machines

- **Domain-specific programming environments trade-off generality to achieve productivity, performance, and portability**
  - Case studies today: Liszt, Halide, OpenGL (see bonus slides)
  - Common trait: languages provide abstractions that make dependencies known
    - Understanding dependencies is necessary but not sufficient: need domain restrictions and domain knowledge for system to synthesize efficient implementations
Bonus slides!

DSL Example 3:

OpenGL: a domain-specific system for 3D rendering
OpenGL graphics pipeline

- Key abstraction: the graphics pipeline
- Graphics pipeline defines a basic program structure and data flows
- Programmable stages (red boxes): programmer fills in the body of the “for all” loops
  (pipeline stage executes “for all” primitives in input stream)

1. Vertices in 3D space (provided by application)
   - Vertex stream
   - Triangle stream
     - Vertex Processing
       - Fragment Generation (Rasterization)
         - Fragment stream
           - Fragment Processing
             - Pixel Blend Operations
               - Output image (pixels)

1. Vertices in 3D space
2. Triangles positioned on screen
3. “Fragments” (one fragment per each covered pixel per triangle)
4. Shaded fragments
Fragment “shader” program

HLSL shader program: defines behavior of fragment processing stage
Executes once per pixel covered by each triangle

Input: a “fragment”: information about the triangle at the pixel
Output: RGBA color (float4 datatype)

```cpp
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv) {
    float3 kd;
    kd = myTex.sample(mySamp, uv);
    kd *= clamp(dot(lightDir, norm), 0.0, 1.0);
    return float4(kd, 1.0);
}
```

Productivity:
- SPMD program: no explicit parallelism
- Implicit parallelism: programmer writes no loops over fragments (think of shader as a loop body)
- Code runs independently for each input fragment (no loops = impossible to express a loop dependency)

Performance:
- SPMD program compiles to wide SIMD processing on GPU
- Work for many fragments dynamically balanced onto GPU cores
- Performance Portability:
  - Scales to GPUs with different # of cores
  - SPMD abstraction compiles to different SIMD widths (NVIDIA=32, AMD=64, Intel=?)
Special language primitive for texture mapping

```cpp
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv) {
    float3 kd;
    kd = myTex.sample(mySamp, uv);
    kd *= clamp(dot(lightDir, norm), 0.0, 1.0);
    return float4(kd, 1.0);
}
```

Intuitive abstraction: represents a texture lookup like an array access with a 2D floating point index.

Texture fetch semantics: sample from `myTex` at coordinate `uv` and filter using scheme (e.g., bilinear filtering) defined by `mySamp`.

Result of mapping texture onto plane, viewed with perspective

`myTex`: 
NxN texture buffer

`uv = (0.3, 0.5)`
Texture mapping is expensive (and performance critical)

- Texture mapping is a filtering operation (more than an array lookup: see 15-462)
  - If implemented in software: ~ 50 instructions, multiple conditionals
  - Read at least 8 values from texture map, blend them together
    - Unpredictable data access, little temporal locality

- Typical shader program performs multiple texture lookups

- Texture mapping is one of the most computationally demanding AND bandwidth intensive aspects of the graphics pipeline
  - Resources for texturing must run near 100% efficiency
  - Not surprising it is encapsulated in its own primitive
Performance: texture mapping

- Highly multi-threaded cores hide latency of memory access (texture primitive = source of long memory stalls is explicit in programming model)
- Fixed-function HW to perform texture mapping math
- Special-cache designs to capture reuse, exploit read-only access to texture data
Performance: global application orchestration

Parallel work:
- Hundreds of thousands of triangles
- Millions of fragments to shade
- Millions of shaded fragments to blend into output image

Efficiently scheduling all this parallel work onto the GPU’s heterogeneous pool of resources (while also respecting the ordering requirements of the OpenGL programming model) is challenging.

Each GPU vendor uses its own custom strategy (high-level abstraction allows for different implementations)
OpenGL summary

- **Productivity:**
  - High-level, intuitive abstractions (taught to undergrads in intro graphics class)
  - Application implements kernels for triangles, vertices, and fragments
  - Specific primitives for key functions like texture mapping

- **Portability**
  - Runs across wide range of GPUs: low-end integrated, high-end discrete, mobile
  - Has allowed significant hardware innovation without impacting programmer

- **High-Performance**
  - Abstractions designed to map efficiently to hardware
    (proposed new features disallowed if they do not!)
  - Encapsulating expensive operations as unique pipeline stages or built-in functions facilitates fixed-function implementations (texture, rasterization, frame-buffer blend)
  - Utilize domain-knowledge in optimizing performance / mapping to hardware
    - Skip unnecessary work, e.g., if a triangle it is determined to be behind another, don’t generate and shade its fragments
    - Non-overlapping fragments are independent despite ordering constraint
    - Interstage queues/buffers are sized based on expected triangle sizes
    - Use pipeline structure to make good scheduling decisions, set work priorities