Lecture 15:

Domain-Specific Programming Systems

Parallel Computer Architecture and Programming

CMU / 清华大学, Summer 2017

Slide acknowledgments:
Pat Hanrahan, Zach Devito (Stanford)
Jonathan Ragan-Kelley (MIT, Stanford)
Course themes:

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

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

**Techniques discussed:**
- Exploiting parallelism in applications
- Exploiting locality in applications
- Leveraging hardware specialization (last time)
Claim: most software uses modern hardware resources inefficiently

- Consider a piece of sequential C code
  - Let’s call 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. ??
Code performance: relative to C (single core)

GCC -O3 (no manual vector optimizations)

- **NBody**
- **Mandlebrot**
- **Tree Alloc/Delloc**
- **Power method (compute eigenvalue)**

Data from: The Computer Language Benchmarks Game: http://shootout.alioth.debian.org
Recall: even good single-threaded C code is inefficient on a modern multi-core CPU

Recall Assignment 1’s Mandelbrot program
Consider execution on this laptop: quad-core, Intel Core i7, AVX...

Single core, with AVX vector instructions: 5.8x speedup over C code
Multi-core + hyper-threading + AVX instructions: ~30-40x speedup

Conclusion: basic C implementation compiled with -O3 leaves a lot of performance on the table
Need for efficiency motivates heterogeneous parallelism

Why specialize hardware? To maximize compute capability given constraints on chip area, energy consumption. Result: amazingly high compute capability in a wide range of devices!

Integrated CPU + GPU

CPU+data-parallel accelerator

Mobile system-on-a-chip: CPU+GPU+media processing

Why specialize hardware? To maximize compute capability given constraints on chip area, energy consumption. Result: amazingly high compute capability in a wide range of devices!
Hardware diversity (needed for efficiency) presents a huge challenge to programmers

- Different machines have very different performance characteristics (different numbers/types of cores, different specialized cores, etc.)

- 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 in one machine: coherent shared memory via fast on-chip network
  - Hybrid CPU+GPU multi-core: incoherent (potentially) shared memory
  - Across racks: distributed memory, multi-stage network
Different programming models emerge to abstract different hardware characteristics

- **Within a core:** SIMD, multi-threading, atomic instructions
  - Abstractions: threads, SPMD programming (ISPC, CUDA, OpenCL, Metal)

- **Across cores:** coherent shared memory via fast on-chip network
  - Abstractions: threads, OpenMP pragma’s, Cilk, TBB

- **Hybrid CPU+GPU multi-core:** incoherent (potentially) shared memory
  - Abstractions: CUDA, OpenCL

- **Across machines:** distributed memory
  - Abstractions: message passing (MPI, Go, Spark, Legion, Charm++)

*Credit: Pat Hanrahan*
Hardware diversity (needed for efficiency) presents a huge challenge (for software portability)

- To be efficient, software must be optimized for the characteristics of target hardware
  - Difficult even in the case of one level of one machine
  - Combinatorial complexity of optimizations when considering a complex machine, or different machines

- Result: loss of software portability
  - All the hard work you do for one parallel computer often needs to get redone for a new type of computer

Credit: Pat Hanrahan
Open computer science question:

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

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
Successful programming languages

Here: definition of success = widely used

High Performance
(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

High Performance (software is scalable and efficient)

Domain-specific languages (DSL) and programming frameworks

Productivity (ease of development)

Completeness (applicable to most problems we want to write a program for)

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: for scientific computing on meshes

2. Halide: for image processing

What are other domain specific languages? (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/
What a Liszt program does

A Liszt program is run on a mesh
A Liszt program defines, and computes the value of, fields defined on the mesh

Position is a field defined at each mesh vertex. The field’s value is represented by a 3-vector.

```
val Position = FieldWithConst[Vertex,Float3](0.f, 0.f, 0.f)
val Temperature = FieldWithConst[Vertex,Float](0.f)
val Flux = FieldWithConst[Vertex,Float](0.f)
val JacobiStep = FieldWithConst[Vertex,Float](0.f)
```

Color key:
- Fields
- Mesh entity

Side note:
Fields are a higher-kind type
(special function that maps a type to a new type)
Liszt program: heat conduction on mesh

Program computes the value of fields defined on meshes

```liszt
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
}
```

Color key:
- **Fields**
- **Mesh**
- **Topology functions**
- **Iteration over set**

Given edge, loop body accesses/modifies field values at adjacent mesh vertices
Liszt's topological operators

Used to access mesh elements relative to some input vertex, edge, face, etc. Topological operators are the only way to access mesh data in a Liszt program. Notice how many operators return sets (e.g., “all edges of this face”)

---

**BoundarySet**

\[ \text{BoundarySet}^1[ME \leftarrow \text{MeshElement}](\text{name : String}) : \text{Set}[\text{ME}] \]

- `vertices(e : Mesh) : Set[Vertex]`
- `cells(e : Mesh) : Set[Cell]`
- `edges(e : Mesh) : Set[Edge]`
- `faces(e : Mesh) : Set[Face]`

- `vertices(e : Vertex) : Set[Vertex]`
- `cells(e : Vertex) : Set[Cell]`
- `edges(e : Vertex) : Set[Edge]`
- `faces(e : Vertex) : Set[Face]`

- `vertices(e : Edge) : Set[Vertex]`
- `facesCCW^2(e : Edge) : Set[Face]`
- `cells(e : Edge) : Set[Cell]`
- `head(e : Edge) : Vertex`
- `tail(e : Edge) : Vertex`
- `flip^4(e : Edge) : Edge`
- `towards^5(e : Edge, t : Vertex) : Edge`

- `cells(e : Cell) : Set[Cell]`
- `vertices(e : Cell) : Set[Vertex]`
- `faces(e : Cell) : Set[Face]`
- `edges(e : Cell) : Set[Edge]`

- `cells(e : Face) : Set[Cell]`
- `edgesCCW^2(e : Face) : Set[Edge]`
- `vertices(e : Face) : Set[Vertex]`
- `inside^3(e : Face) : Cell`
- `outside^3(e : Face) : Cell`
- `flip^4(e : Face) : Face`
- `towards^5(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 target machine

Well, that’s interesting. I write a program, and the compiler decides what data structure it should use based on what operations my code performs.
Compiling to parallel computers

Recall challenges you have faced in your assignments

1. Identify potential parallelism (lack of dependencies)
2. Identify data locality
3. Reason about what synchronization is required

Now consider how to automate this process if you were implementing a Liszt compiler.
Key: determining program dependencies

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

2. Identify data locality
   - Partition data based on dependencies

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

Given general purpose programming languages (like C or Java programs), compilers are unable to infer dependencies at global scale:

Consider: \( a[f(i)] += b[i] \);
(must execute \( f(i) \) to know if dependency exists across loop iterations \( i \))
Liszt is constrained to allow dependency analysis

Liszt infers “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 D and F
Restrict language for dependency analysis

Language restrictions:

- Mesh elements are only accessed through built-in topological functions:

  \texttt{cells(mesh)}, \ldots

- Single static assignment: (immutable values)

  \texttt{val v1 = head(e)}

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

  \texttt{Pressure(v)}

- No recursive functions

Restrictions allow compiler to automatically infer stencil for a loop iteration
Portable parallelism: compiler uses knowledge of dependencies to implement different parallel execution strategies

I’ll discuss two strategies…

Strategy 1: mesh partitioning for a cluster

Strategy 2: mesh coloring for GPU
Imagine compiling a Lizst program to a cluster (multiple nodes, distributed address space)

How might Lizst distribute a graph across these nodes?
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 to implement topological operators.
Imagine compiling a Lizst program to a GPU (single address space, many tiny threads)
GPU implementation: parallel reductions

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

Edges (each edge assigned to 1 CUDA thread)

<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>
</table>

Flux field values (stored per vertex)

```
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

Edges (each edge assigned to 1 CUDA thread)

Flux field values (per vertex)

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

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

Threads (each edge assigned to 1 CUDA thread)

<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>
</table>

Flux field values (per vertex)

“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.
Cluster performance of Lizst program

256 nodes, 8 cores per node (message-passing implemented using MPI)

Important: performance portability!
Same Liszt program also runs with high efficiency on GPU (results not shown)
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 large cluster of CPUs and GPUs (and combinations thereof!)

- **High performance**
  - Language is constrained to allow compiler to track dependencies
  - Used for locality-aware partitioning (distributed memory implementation)
  - Used for graph coloring to avoid sync (GPU implementation)
  - Compiler chooses different parallelization strategies for different platforms
  - System can customize mesh representation based on application and platform (e.g., don’t store edge pointers if code doesn’t need it, choose struct of arrays vs. array of structs for per-vertex fields)
Example 2:
Halide: a domain-specific language for image processing

Jonathan Ragan-Kelley, Andrew Adams et al.
[SIGGRAPH 2012, PLDI 13]
Halide used in practice

- Halide used to implement Google Pixel Photos app
- Halide code used to process images uploaded to Google Photos
A quick tutorial on high-performance image processing
What does this C code do?

```c
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1.f/9, 1.f/9, 1.f/9, 
                  1.f/9, 1.f/9, 1.f/9, 
                  1.f/9, 1.f/9, 1.f/9};

for (int j=0; j<HEIGHT; j++) {
    for (int i=0; i<WIDTH; i++) {
        float tmp = 0.f;
        for (int jj=0; jj<3; jj++)
            for (int ii=0; ii<3; ii++)
                tmp += input[(j+jj)*(WIDTH+2) + (i+ii)] * weights[jj*3 + ii];
        output[j*WIDTH + i] = tmp;
    }
}
```
3x3 box blur

(Zoom view)
3x3 image blur

```c
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1.f/9, 1.f/9, 1.f/9,
                   1.f/9, 1.f/9, 1.f/9,
                   1.f/9, 1.f/9, 1.f/9};

for (int j=0; j<HEIGHT; j++) {
    for (int i=0; i<WIDTH; i++) {
        float tmp = 0.f;
        for (int jj=0; jj<3; jj++)
            for (int ii=0; ii<3; ii++)
                tmp += input[(j+jj)*(WIDTH+2) + (i+ii)] * weights[jj*3 + ii];
        output[j*WIDTH + i] = tmp;
    }
}
```

Total work per image $= 9 \times WIDTH \times HEIGHT$

For $N \times N$ filter: $N^2 \times WIDTH \times HEIGHT$
Two-pass blur

A 2D separable filter (such as a box filter) can be evaluated via two 1D filtering operations

Note: I’ve exaggerated the blur for illustration (the end result is 30x30 blur, not 3x3)
Two-pass 3x3 blur

```c
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1.f/3, 1.f/3, 1.f/3};

for (int j=0; j<(HEIGHT+2); j++)
    for (int i=0; i<WIDTH; i++)
        {
            float tmp = 0.f;
            for (int ii=0; ii<3; ii++)
                tmp += input[j*(WIDTH+2) + i+ii] * weights[ii];
            tmp_buf[j*WIDTH + i] = tmp;
        }

for (int j=0; j<HEIGHT; j++)
    for (int i=0; i<WIDTH; i++)
        {
            float tmp = 0.f;
            for (int jj=0; jj<3; jj++)
                tmp += tmp_buf[(j+jj)*WIDTH + i] * weights[jj];
            output[j*WIDTH + i] = tmp;
        }
```

Total work per image = 6 x WIDTH x HEIGHT
For NxN filter: 2N x WIDTH x HEIGHT
WIDTH x HEIGHT extra storage
2X lower arithmetic intensity than 2D blur
Two-pass image blur: locality

```c
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1.f/3, 1.f/3, 1.f/3};

for (int j=0; j<(HEIGHT+2); j++)
   for (int i=0; i<WIDTH; i++)
      {float tmp = 0.f;
       for (int ii=0; ii<3; ii++)
          tmp += input[j*(WIDTH+2) + i+ii] * weights[ii];
       tmp_buf[j*WIDTH + i] = tmp;
      }

for (int j=0; j<HEIGHT; j++)
   for (int i=0; i<WIDTH; i++)
      {float tmp = 0.f;
       for (int jj=0; jj<3; jj++)
          tmp += tmp_buf[(j+jj)*WIDTH + i] * weights[jj];
       output[j*WIDTH + i] = tmp;
      }
```

Intrinsic bandwidth requirements of blur algorithm:
Application must read each element of input image
and must write each element of output image.

Data from input reused three times. (immediately reused in next
two i-loop iterations after first load, never loaded again.)
- Perfect cache behavior: never load required data more than once
- Perfect use of cache lines (don’t load unnecessary data into cache)

Data from tmp_buf reused three times (but three rows of image data are accessed in between)
- Never load required data more than once… if
  cache has capacity for three rows of image
- Perfect use of cache lines (don’t load unnecessary data into cache)

Two pass: loads/stores to tmp_buf are overhead (this memory traffic
is an artifact of the two-pass implementation: it is not intrinsic to
computation being performed)
Two-pass image blur, “chunked” (version 1)

```c
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * 3];
float output[WIDTH * HEIGHT];
float weights[] = {1.f/3, 1.f/3, 1.f/3};

for (int j=0; j<HEIGHT; j++) {
    for (int j2=0; j2<3; j2++)
        for (int i=0; i<WIDTH; i++) {
            float tmp = 0.f;
            for (int ii=0; ii<3; ii++)
                tmp += input[(j+j2)*(WIDTH+2) + i+ii] * weights[ii];
            tmp_buf[j2*WIDTH + i] = tmp;
        }
    for (int i=0; i<WIDTH; i++) {
        float tmp = 0.f;
        for (int jj=0; jj<3; jj++)
            tmp += tmp_buf[jj*WIDTH + i] * weights[jj];
        output[j*WIDTH + i] = tmp;
    }
}
```

- **input** $(W+2) \times (H+2)$  
- **tmp_buf** $(W \times 3)$  
- **output** $W \times H$

Only 3 rows of intermediate buffer need to be allocated

Produce 3 rows of tmp_buf (only what’s needed for one row of output)

Combine them together to get one row of output

**Total work per row of output:**
- step 1: $3 \times 3 \times WIDTH$ work  
- step 2: $3 \times WIDTH$ work

**Total work per image** $= 12 \times WIDTH \times HEIGHT$

Loads from tmp_buffer are cached (assuming tmp_buffer fits in cache)
Two-pass image blur, “chunked” (version 2)

```c
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (CHUNK_SIZE+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1.f/3, 1.f/3, 1.f/3};

for (int j=0; j<HEIGHT; j+CHUNK_SIZE) {
    for (int j2=0; j2<CHUNK_SIZE+2; j2++)
        for (int i=0; i<WIDTH; i++)
            float tmp = 0.f;
        for (int ii=0; ii<3; ii++)
            tmp += input[(j+j2)*(WIDTH+2) + i+ii] * weights[ii];
        tmp_buf[j2*WIDTH + i] = tmp;
    for (int j2=0; j2<CHUNK_SIZE; j2++)
        for (int i=0; i<WIDTH; i++)
            float tmp = 0.f;
        for (int jj=0; jj<3; jj++)
            tmp += tmp_buf[(j2+jj)*WIDTH + i] * weights[jj];
        output[(j+j2)*WIDTH + i] = tmp;
}
```

Sized so entire buffer fits in cache (capture all producer-consumer locality)

Produce enough rows of tmp_buf to produce a CHUNK_SIZE number of rows of output

Produce CHUNK_SIZE rows of output

Total work per chunk of output:
(assume CHUNK_SIZE = 16)
- Step 1: 18 x 3 x WIDTH work
- Step 2: 16 x 3 x WIDTH work
Total work per image: \((34/16) \times 3 \times WIDTH \times HEIGHT\) = 6.4 x WIDTH x HEIGHT

Trends to ideal value of 6 x WIDTH x HEIGHT as CHUNK_SIZE is increased!
Still not done

- We have not parallelized loops for multi-core execution
- We have not used SIMD instructions to execute loops bodies
- Other basic optimizations: loop unrolling, etc...
Optimized C++ code: 3x3 image blur

Good: ~10x faster on a quad-core CPU than my original two-pass code
Bad: specific to SSE (not AVX2), CPU-code only, hard to tell what is going on at all!

```c
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+1));
                    c = _mm_load_si128((~m128i*)(inPtr));
                    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 language

Simple language embedded in C++ for describing sequences of image processing operations (“image processing pipelines”)

```
Var x, y;
Func blurx, blury, out;
Image<uint8_t> in = load_image("myimage.jpg");

// perform 3x3 box blur in two-passes
blurx(x,y) = (in(x-1,y) + in(x,y) + in(x,y)) / 3.f;
blury(x,y) = (blurx(x,y-1) + blurx(x,y+1) + blurx(x,y+1)) / 3.f;

// brighten blurred result by 25%, then clamp
out(x,y) = min(blury(x,y) * 1.25f, 255);

// execute pipeline on domain of size 800x600
Image<uint8_t> result = out.realize(800, 600);
```

- Halide function: an infinite (but discrete) set of values
- Halide expression: a side-effect free expression describes how to compute a function’s value at a point in it’s domain in terms of the values of other functions.

[Halide, Ragan-Kelley 2012]
Key aspects of Halide’s design

- Local “pointwise” view of expressing algorithms
- Language is constrained so that iteration over domain points is implicit (no explicit loops in Halide)
  - Halide is declarative. It does not define order of iteration, or what values in domain or stored!
  - It only defines what operations are needed to compute these values.

Var x, y;
Func blurx, out;
Image<uint8_t> in = load_image("myimage.jpg");

// perform 3x3 box blur in two-passes
blurx(x,y) = (in(x-1,y) + in(x,y) + in(x,y)) / 3.f;
out(x,y) = (blurx(x,y-1) + blurx(x,y+1) + blurx(x,y+1)) / 3.f;

// execute pipeline on domain of size 800x600
Image<uint8_t> result = our.realize(800, 600);
Real-world image processing pipelines feature complex sequences of functions.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Number of Halide functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two-pass blur</td>
<td>2</td>
</tr>
<tr>
<td>Unsharp mask</td>
<td>9</td>
</tr>
<tr>
<td>Harris Corner detection</td>
<td>13</td>
</tr>
<tr>
<td>Camera RAW processing</td>
<td>30</td>
</tr>
<tr>
<td>Non-local means denoising</td>
<td>13</td>
</tr>
<tr>
<td>Max-brightness filter</td>
<td>9</td>
</tr>
<tr>
<td>Multi-scale interpolation</td>
<td>52</td>
</tr>
<tr>
<td>Local-laplacian filter</td>
<td>103</td>
</tr>
<tr>
<td>Synthetic depth-of-field</td>
<td>74</td>
</tr>
<tr>
<td>Bilateral filter</td>
<td>8</td>
</tr>
<tr>
<td>Histogram equalization</td>
<td>7</td>
</tr>
<tr>
<td>VGG-16 deep network eval</td>
<td>64</td>
</tr>
</tbody>
</table>

Real-world production applications may feature hundreds to thousands of functions!

Google HDR+ pipeline: over 2000 Halide functions.
Key aspect in the design of any system:

Choosing the “right” representations for the job

Now the job is not expressing an image processing computation, but generating an efficient implementation of a specific Halide program.
A second set of representations for “scheduling”

Func blurx, out;
Var x, y, xi, yi;
Image<
> in = load_image(“myimage.jpg”);

// the “algorithm description” (declaration of what to do)
blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;
out(x,y) = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;

// “the schedule” (how to do it)
out.tile(x, y, xi, yi, 256, 32).vectorize(xi,8).parallel(y);
blurx.compute_at(x).vectorize(x, 8);

When evaluating out, 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 elements blurx on demand for each tile of output.
Vectorize the x (innermost) loop

// execute pipeline on domain of size 1024x1024
Image<
> result = out.realize(1024, 1024);

Scheduling primitives allow the programmer to specify a global “sketch” of how to schedule the algorithm onto a parallel machine, but leave the details of emitting the low-level platform-specific code to the Halide compiler.
Primitives for iterating over domains

Specify both order and how to parallelize (multi-thread, vectorize via SIMD instr)

2D blocked iteration order
Specifying loop iteration order and parallelism

\[
\text{blurx}(x, y) = \frac{\text{in}(x-1, y) + \text{in}(x, y) + \text{in}(x+1, y)}{3.0f};
\]
\[
\text{out}(x, y) = \frac{\text{blurx}(x, y-1) + \text{blurx}(x, y) + \text{blurx}(x, y+1)}{3.0f};
\]

Given this schedule for the function “out”...

\[
\text{out}.\text{tile}(x, y, xi, yi, 256, 32).\text{vectorize}(xi, 8).\text{parallel}(y);
\]

Halide compiler will generate this parallel, vectorized loop nest for computing elements of \text{out}...

\[
\text{for } y=0 \text{ to } \text{num\_tiles\_y:} \quad \text{// parallelize this loop over multiple threads}
\]
\[
\text{for } x=0 \text{ to } \text{num\_tiles\_x:}
\]
\[
\text{for } yi=0 \text{ to } 32:
\]
\[
\text{for } xi=0 \text{ to } 256: \quad \text{// vectorize this loop with SIMD instructions}
\]
\[
\text{idx\_x} = x*256+xi;
\]
\[
\text{idx\_y} = y*32+yi
\]
\[
\text{out}(\text{idx\_y}, \text{idx\_y}) = ...
\]
Primitives for how to interleave producer/consumer processing

\[ \text{blurx}(x,y) = \frac{(\text{in}(x-1, y) + \text{in}(x,y) + \text{in}(x+1,y))}{3.0f}; \]
\[ \text{out}(x,y) = \frac{(\text{blurx}(x,y-1) + \text{blurx}(x,y) + \text{blurx}(x,y+1))}{3.0f}; \]
\[ \text{out}.\text{tile}(x, y, x_i, y_i, 256, 32); \]

\[ \text{blurx}.\text{compute\_root}(); \]

Do not compute blurx within out's loop nest.
Compute all of blurx, then all of out

allocate buffer for all of blur(x,y)
for y=0 to HEIGHT:
  for x=0 to WIDTH:
    blurx(x,y) = …

for y=0 to num_tiles_y:
  for x=0 to num_tiles_x:
    for yi=0 to 32:
      for xi=0 to 256:
        idx_x = x*256+xi;
        idx_y = y*32+yi
        out(idx_y, idx_y) = …

all of blurx is computed here

values of blurx consumed here
Primitives for how to interleave producer/consumer processing

\[
\text{blurx}(x,y) = \frac{\text{in}(x-1, y) + \text{in}(x,y) + \text{in}(x+1,y)}{3.0f};
\]
\[
\text{out}(x,y) = \frac{\text{blurx}(x,y-1) + \text{blurx}(x,y) + \text{blurx}(x,y+1)}{3.0f};
\]
\[
\text{out}.\text{tile}(x, y, x_i, y_i, 256, 32);
\]

\[
\text{blurx}.\text{compute}_\text{at}(x_i);
\]

Compute necessary elements of blurx within out’s xi loop nest

\[
\text{for } y=0 \text{ to } \text{num}_\text{tiles}_y:
\]
\[
\text{for } x=0 \text{ to } \text{num}_\text{tiles}_x:
\]
\[
\text{for } y_i=0 \text{ to } 32:
\]
\[
\text{for } x_i=0 \text{ to } 256:
\]
\[
\text{id}_x = x \times 256 + x_i;
\]
\[
\text{id}_y = y \times 32 + y_i
\]

allocate 3-element buffer for blurx
// compute 3 elements of blurx needed for out(idx_x, idx_y) here

\[
\text{out}(\text{id}_y, \text{id}_y) = \ldots
\]
Primitives for how to interleave producer/consumer processing

\[
\begin{align*}
\text{blurx}(x, y) &= \frac{(\text{in}(x-1, y) + \text{in}(x, y) + \text{in}(x+1, y))}{3.0f}; \\
\text{out}(x, y) &= \frac{\text{blurx}(x, y-1) + \text{blurx}(x, y) + \text{blurx}(x, y+1))}{3.0f}; \\
\text{out}.tile(x, y, xi, yi, 256, 32); \\
\end{align*}
\]

\text{blurx}.compute_at(x);

Compute necessary elements of blurx within out's x loop nest (all necessary elements for one tile of out)

for y=0 to num_tiles_y:
  for x=0 to num_tiles_x:

    allocate 258x34 buffer for tile blurx
    for yi=0 to 32+2:
      for xi=0 to 256+2:
        blur(xi,yi) = // compute blurx from in

    for yi=0 to 32:
      for xi=0 to 256:
        idx_x = x*256+xi;
        idx_y = y*32+yi
        out(idx_y, idx_y) = ...

  tile of blurx is computed here

  tile of blurx is consumed here
Halide: two domain-specific co-languages

- Functional primitives for describing image processing operations
- Additional primitives for describing schedules
- Design principle: separate “algorithm specification” from schedule
  - Programmer’s responsibility: provide a high-performance schedule
  - Compiler’s responsibility: carry out mechanical process of generating threads, SIMD instructions, managing buffers, etc.
  - Result: enable programmer to rapidly exploration of space of schedules (“tile these loops”, “vectorize this loop”, “parallelize this loop across cores”)

- Application domain scope:
  - All computation on regular N-D coordinate spaces
  - Only feed-forward pipelines (includes special support for reductions and fixed recursion depth)
  - All dependencies inferable by compiler
Early Halide results

- Camera RAW processing pipeline
  (Convert RAW sensor data to RGB image)
  - Original: 463 lines of hand-tuned ARM NEON 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

[Ragan-Kelley 2012]
Stepping back: what is Halide?

- Halide is a DSL for helping expert developers optimize image processing code more rapidly
  - Halide does not decide how to optimize a program for a novice programmer
  - Halide provides primitives for a programmer (that has strong knowledge of code optimization, such as a 15-418 student) to rapidly express what optimizations the system should apply
  - Halide compiler carries out the nitty-gritty of mapping that strategy to a machine
Automatically generating Halide schedules

- Problem: it turned out that very few programmers have the ability to write good Halide schedules
  - 80+ programmers at Google write Halide
  - Very small number trusted to write schedules

- Recent work: compiler analyzes the Halide program to automatically generate efficient schedules for the programmer
  [Mullapudi 2016]
### Autoscheduler performs comparably to experts

#### Performance relative to schedules authored by experts

(6 core Xeon CPU)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Auto scheduler</th>
<th>Baseline</th>
<th>Relative Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bilateral grid</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blur</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Camera pipe</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Convolution layer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Harris corner</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Histogram equal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mscale interpolate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lens blur</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local laplacian</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matrix multiply</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max filter</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-local means</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unsharp mask</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VGG-16 evaluation</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Performance relative to schedules authored by experts on 8 of the 14 benchmarks performance within 10% of experts or better.

Baseline schedules exploit multi-core/vector parallelism and pointwise inlining but no global locality optimizations.
Halide auto scheduler vs. experts
(graphics plot expert code performance as they work to manually tune the code)

Non-local means denoising

- Throughput vs. Time (min)

Lens blur

- Throughput vs. Time (min)

Max filter

- Throughput vs. Time (min)

Legend:
- Green: Auto scheduler
- Gray: Dillon
- Red: Andrew
Darkroom/Rigel

Goal: directly synthesize FPGA implementation of image processing pipelines from a high-level description (a constrained “Halide-like” language)

```
bx = im(x,y)  
(I(x-1,y) + I(x,y) + I(x+1,y))/3  
end
by = im(x,y)  
(bx(x,y-1) + bx(x,y) + bx(x,y+1))/3  
end
sharpened = im(x,y)  
I(x,y) + 0.1*  
(I(x,y) - by(x,y))  
end  
```

Goal: very-high efficiency image processing
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 Ligra (DSLs for describing operations on graphs)

DSL for defining deep neural networks and training/inference computations on those networks

Language for real-time 3D graphics

Numerical computing

Ongoing efforts in many domains...

Languages for physical simulation: Simit [MIT], Ebb [Stanford]
Opt: a language for non-linear least squares optimization [Stanford]
Elements of good domain-specific programming system design
#1: good systems identify the most important cases for a domain, and provide most benefit in these situations

- Structure of code mimics the natural structure of problems in the domain
  - “Code should reflect how domain practitioners think about the problem”
  - Halide: pixel-wise view of filters: programmer provides expression for how pixel(x,y) is computed (“average of its neighbors”)
  - Graph/mesh processing algorithms designed in terms of per-vertex operations

- Productive expression of applications: common operations are easy and intuitive to express

- Efficient implementation: the most important optimizations used in the domain are performed by the system for the programmer
  - My experience: a parallel programming system that provides “convenient” syntax but prevent using best-known implementation strategies will almost always fail
#2: good systems are usually simple systems

- They have a small number of key primitives and operations
  - Ligra (next lecture): only two operations! (vertexmap and edgemap)
  - GraphLab (next lecture): run computation per vertex, trigger new work by signaling
    - But GraphLab gets messy with all the scheduling options
  - Halide: only a few scheduling primitives
  - Hadoop: map + reduce

- Compiler/runtime focuses on optimizing these primitives
  - Provide parallel implementations, utilize appropriate specialized hardware, etc.

- Common question that good architects ask: “do we really need that?” (can this concept be reduced to a primitive we already have?)
  - For every domain-specific primitive in the system: there better be a strong performance or expressivity justification for its existence
#3: good primitives compose

- Composition of primitives allows for wide application scope within the domain
  - e.g., frameworks discussed today support a wide variety of graph algorithms
  - Halide’s loop ordering + loop interleaving schedule primitives allow for expression of wide range of schedules

- Composition often allows optimizations generalize
  - If system can optimize A and optimize B, then it can optimize programs that combine A and B

- Sign of a good design:
  - System is later used for applications original designers never anticipated

- Sign that a feature **should not** be added (or added in a different way):
  - The new feature does not compose with all existing features in the system
Summary

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

- Most software uses small fraction 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
  - Compiler/runtime leverages explicit dependencies, language restrictions, and domain knowledge to synthesize efficient implementations