Lecture 5:
SIMD Parallelism
For Rendering

Image Synthesis
Stanford CS348b, Spring 2014
Three Forms of Parallelism in Modern CPUs

- Multi-core parallelism
  - Different processing cores run different instruction streams
  - (April 25 lecture)

- Instruction-level parallelism
  - Multiple execution units execute multiple instructions in parallel
  - Transparent to the programmer

- SIMD parallelism
  - Single instruction stream expresses vector computation
Why multi-core and SIMD? Why now?

- Moore’s law continues for now
  - Only a statement about transistor size, now power
- Until ~2000, more transistors $\rightarrow$ better single-thread perf.
  - See “The Slow Winter” reading
- Since ~2000, most perf. improvement opportunities through explicit parallelism
  - Impacts software for the first time
- Today, ~all computing is power limited
  - Can’t light up all the transistors Moore’s law gives us
- Multi-core, SIMD provide more power-efficient computation
Pre-SIMD Era CPU (ca. 2000)

- Fetch/Decode
- ALU
- Execution Context (Registers, etc.)
- Cache
Example: Scalar Execution

```c
void cube8(float *array) {
    for (int i = 0; i < 8; ++i) {
        float v = array[i];
        v = v * v * v;
        array[i] = v;
    }
}
```

Instruction stream processes one array element at a time using scalar instructions on scalar registers (e.g., 32-bit floats)

```assembly
label:
    load   r0, addr[r1]
    mul    r2, r0, r0
    mul    r2, r2, r0
    store  addr[r1], r2
    # prepare for next iteration
    # jump to label if need more iterations
```
SIMD Hardware: Motivation

Amortize cost/complexity of managing an instruction stream across multiple ALUs

**SIMD processing**

Single instruction, multiple data

Same instruction broadcast to all ALUs  
Executed in parallel on all ALUs

Diagram:
- Fetch/Decode
- ALUs (ALU 1 to ALU 8)
- Execution Context (Registers, etc.)
- Cache
Example: SIMD Execution

```
void cube8(float *array) {
    for (int i = 0; i < 8; ++i) {
        float v = array[i];
        v = v * v * v;
        array[i] = v;
    }
}
```

Instruction stream processes 8 array elements at a time using SIMD instructions with SIMD registers
SIMD on Various Processors

- Intel Core i3/i5/i7: 8-wide float / 32-bit int SIMD
  - 4 cores x 8-wide SIMD: 32x perf. difference available if parallelism is used effectively

- GPUs: 16 to 64-wide SIMD on each processing core
- Pixar Image Computer (RIP)
Expressing SIMD: Auto-Vectorization

- **cc -O2 -fvectorize ...**

- **Advantages:**
  - Maintain serial program expression
  - Trivial for programmer

- **Disadvantages:**
  - Auto-vectorization algorithms are finicky in practice
  - Little performance portability across compilers
  - Poor performance transparency
Expressing SIMD: Explicit Vectors

- Explicit vector types in language or library
- Programmer directly operates on vector types
- Serial computation model
- Advantages:
  - Easy for compiler
  - Cross-element operations in vectors are clear

```c
void cube(float *array, int n) {
    assert((n % 8) == 0);
    for (int i = 0; i < n; i += 8) {
        vec<float, 8> v = load<8>(&array[i]);
        v = v * v * v;
        store(&array[i], v);
    }
}
```
A Small SIMD Vector Library API

- For the following, we’ll assume we have a vector library that provides this interface
- Almost all of these operations have a 1:1 mapping to hardware instructions
- Implementation via ungainly compiler intrinsics

```
template <typename T, int width> struct vec {
    vec(T v) { /* smear */ }

    T &operator[](int index); // index >= 0, < width

    vec<T, width> operator+(vec<T, width>) const;
    // ...

    vec<bool, width> operator<(vec<T, width>) const;
    // ...
};

vec<T, width> id(); // 0, 1, 2, ...

vec<T, width> select(vec<bool, width>,
    vec<T, width> a, vec<T, width> b);

vec<float, 8> operator*(vec<float, 8> v2) const
    vec<float, 8> ret;
    ret.v = _mm256_mul_ps(v.v, v2.v);
    return ret;
```
Applying SIMD To Rasterization

- Let’s start with triangle setup...
  - Assume we’re provided batches of 8 triangles, structured like so:
  - And now we’d like to set up 8 triangles at once with SIMD
  - First edge equation coefficient:

```cpp
// a0 = -(y2 - y1)
vec<float, 8> a0 = -(batch.t[id()].v[2].y - batch.t[id()].v[1].y);
```

- What do these memory accesses look like?

```cpp
struct Vertex {
    float x, y, z;
    float r, g, b;
};
struct Triangle {
    Vertex v[3];
};
struct TriangleBatch {
    Triangle t[8];
};
```
Array of Structures Memory Layout

```c
struct Vertex {
    float x, y, z;
    float r, g, b;
};

struct Triangle {
    Vertex v[3];
};

struct TriangleBatch {
    Triangle t[8];
};
```

TriangleBatch:

Triangle:

Vertex:
- x y z r g b

```
x y z r g b x y z r g b x y z r g b x y z r g b x y z r g b...
```

```c
batch.t[id()].v[1].y
```
Structure of Arrays Memory Layout

- Consider an alternate definition of a batch of triangles:

  ```cpp
  struct TriangleBatch {
    vec<8, float> x[3], y[3], z[3];
    vec<8, float> r[3], g[3], b[3];
  };
  ```

- Setup expressions are more clear:

  ```cpp
  vec<float, 8> a0 = -(batch.y[2] - batch.y[1]);
  ```

- Loads now hit contiguous memory locations
SIMD Triangle Setup

- Most of the triangle setup logic now looks the same as when we were setting up a single triangle at a time:

```cpp
void Rasterize(TriangleBatch &b) {
    vec<float, 8> area = 0.5f * ((b.x[1] - b.x[0]) * (b.y[2] - b.y[0]) -
                                (b.y[1] - b.y[0]) * (b.x[2] - b.x[0]));
    vec<float, 8> inv2Area = 1.f / (2.f * area);
    vec<bool, 8> backfacing = (area <= 0.);
    // if (backfacing) return;

    vec<float, 8> a0 = -(b.y[2] - b.y[1]), b0 = b.x[2] - b.x[1];
    vec<float, 8> c0 = a0 * -b.x[1] + b0 * -b.y[1];
    vec<bool, 8> inc0 = (a0 > 0) || (a0 == 0 && b0 > 0);
    // a1, b1, c1, inc1, a2, b2, c2, inc2...
}
```
SIMD Triangle Setup

- What about this?

```cpp
vec<bool, 8> backfacing = (area <= 0.);
// if (backfacing) return;
```

- Assume we have some helper functions:

```cpp
bool all(vec<bool, width>);
bool none(vec<bool, width>);
bool any(vec<bool, width> v) { return !all(v) && !none(v); }
```

- We can handle some cases:

```cpp
vec<bool, 8> backfacing = (area <= 0.);
if (all(backfacing)) return;
```

- But still have the underlying issue of what to do about when some but not all are backfacing...
Two Options for Unoccupied SIMD Lanes

- Just don’t worry about it
  - Continue setup as usual
  - Edge equation values computed for backfacing triangles aren’t needed, but they’re free to compute

- Repack vectors to improve SIMD occupancy
  - Trade-off of work to shuffle data around vs. benefits from better SIMD occupancy in subsequent computations...

```cpp
struct TriangleBatch {
  vec<8, float> x[3], y[3], z[3];
  vec<8, float> r[3], g[3], b[3];
};
```
Nested Parallelism / SIMD

- For each triangle:
  - Compute edge function coefficients
  - Determine if back-facing
  - For each front-facing triangle
    - Test candidate samples against triangle

```cpp
vec<bool, 8> backfacing = (area <= 0.);
// ...
for (int i = 0; i < 8; ++i) {
  if (backfacing[i])
    continue;
  float ta0 = a0[i], tb0 = b0[i], tc0 = c0[i];
  bool tinc0 = inc0[i];
  // ta1, tb1, tc1, tinc1, ...
```
SIMD Rasterization: Sample Testing

- Single triangle, SIMD over multiple samples
  - Test one sample in each SIMD lane
- Can consider horizontal spans of samples
SIMD Rasterization: Sample Testing

- Better: square or rectangular **tiles** of samples
  - (Square not possible for all SIMD widths)
SIMD Rasterization: Sample Testing

- Initialize elements of vectors to correspond to samples in a tile
- Very similar to the regular rasterizer from before

```cpp
#define ALIGN(v, a) ((v) + (a) - 1) & ~(a-1)
y0 = ALIGN(y0, 2);
x0 = ALIGN(x0, 4);
vec<float, 8> vy = y0 + vec<float, 8>(0, 0, 0, 0, 1, 1, 1, 1);
vec<float, 8> vx = x0 + vec<float, 8>(0, 1, 2, 3, 0, 1, 2, 3);

for (float y = y0; y < y1; y += 2, vy += 2) {
    float (float x = x0; x < x1; x += 4, vx += 4) {
        vec<float, 8> e0 = a0 * vx + b0 * vy + c0;
        vec<float, 8> e1 = a1 * vx + b1 * vy + c1;
        vec<float, 8> e2 = a2 * vx + b2 * vy + c2;
        vec<bool, 8> outside = ((e0 < 0. || (e0 == 0. && !inc0)) &&
                                ((e1 < 0. || (e1 == 0. && !inc1)) &&
                                 ((e2 < 0. || (e2 == 0. && !inc2)));
        if (all(outside)) continue;
        // update the framebuffer...
```
SIMD Rasterizer Framebuffer Update

- For coherent memory access, arrange the framebuffer tiled in memory:

```
vec<float, 8> depthBuffer[yResolution/2][xResolution/4];
```

First tile of samples

- Then, perform masked update based on inside and z-tests

```plaintext
vec<float, 8> w0 = e0 * inv2Area, w1 = e1 * inv2Area, w2 = e2 * inv2Area;
vec<float, 8> z = w0 * z0 + w1 * z1 + w2 * z2;
vec<bool, 8> mask = !outside & (z < depthBuffer[x][y]);
depthBuffer[x][y] = select(mask, z, depthBuffer[x][y]);
// interpolate r, g, b
// update r, g, b similarly
```
Framebuffer Memory Layout for SIMD

Linear

Tiled 4x2
We can also use SIMD to cull entire tiles worth of samples, performing multiple corner vs. edge tests at the same time.
Full SIMD Tiled Rasterization Algorithm

- For each triangle:
  - Compute edge function coefficients
  - Determine if back-facing
  - For each front-facing triangle
    - Compute which tiles the triangle’s bounding box overlaps
    - For each overlapping tile
      - Perform edge/tile culling test
    - For each active tile
      - For each sample in tile
        - Test sample against triangle
Implicit vs. Explicit Expression of SIMD

- Explicit vectors are one way to express SIMD computation
  - This is what we did for rasterization

- It can also be expressed implicitly with a language that allows describing the computation to do for each data element
  - “For each grid vertex, run this surface shader”

```c
color shade(color Kd, float3 N, float3 L) {
    return Kd * max(0., N.x * L.x + N.y * L.y + N.z * L.z);
}
```
Implicit Expression of SIMD

“For each grid vertex, run this surface shader”

color shade(color Kd, float3 N, float3 L) {
    return Kd * max(0., N.x * L.x + N.y * L.y + N.z * L.z);
}

Decompose grid into groups of 8 vertices; run this code for each one:

struct f3 { vec<float, 8> v[3]; };

f3 shade(f3 Kd, f3 N, f3 L) {
    dot = select(dot < 0., 0., dot);
    f3 ret;
    ret.v[0] = Kd.v[0] * dot;
    return ret;
}
Data-Parallel Shader Execution Model

- Data-parallel computation: perform the same computation over multiple input values
- For shaders:
  - The same computation is performed at each grid point
  - The value computed at each grid point is independent of all other grid points*
- Advantages:
  - Deterministic, deadlock free model of parallel computation
  - Cleanly maps to SIMD execution hardware

* 99% accurate statement; bear with me for now
Memory Layout for SIMD Grid Shading

```c
struct GridSoA {
    float px[MAX_V], py[MAX_V], pz[MAX_V];
    // ...
};
```

Loading data with a structure of arrays (SoA) layout for a group of grid points only requires a vector load from memory.
RenderMan Shader

- Implicit specification of surface texture and appearance

```
surface checker(float Kd = .5, Ka = .1, frequency = 10; color blackcolor = color (0, 0, 0)) {
    float smod = mod (s* frequency, 1),
        tmod = mod (t* frequency, 1);

    if (smod < 0.5) {
        if (tmod < 0.5)
            Ci = Cs;
        else
            Ci = blackcolor;
    } else {
        if (tmod < 0.5)
            Ci = blackcolor;
        else
            Ci = Cs;
    }

    Oi = Os;
    Ci = Oi * Ci *
        (Ka * ambient() +
         Kd * diffuse(faceforward(normalize(N), I)));
}
```

- This also was designed to run on SIMD hardware...
  - Consider each variable to be a vector value...
What about conditional execution?

Time (clocks)

ALU 1  ALU 2  ...  ...  ALU 8

float tmp = exp(x, 5);
tmp *= kMyConst1;
x = tmp + kMyConst2;

float x = A[i];
result[i] = x;

(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')
What about conditional execution?

(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```c
float x = A[i];
if (x > 0) {
    float tmp = exp(x,5);
    tmp *= kMyConst1;
    x = tmp + kMyConst2;
} else {
    float tmp = kMyConst1;
    x = 2.f * tmp;
}
result[i] = x;
```
Mask (discard) output of ALU

Not all ALUs do useful work!
Worst case: 1/8 peak performance

(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
float x = A[i];
if (x > 0) {
    float tmp = exp(x, 5);
    tmp *= kMyConst1;
    x = tmp + kMyConst2;
} else {
    float tmp = kMyConst1;
    x = 2.f * tmp;
}
result[i] = x;
```
After branch: continue at full performance

(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```c
float x = A[i];
if (x > 0) {
    float tmp = exp(x, 5);
    tmp *= kMyConst1;
    x = tmp + kMyConst2;
} else {
    float tmp = kMyConst1;
    x = 2.f * tmp;
}

result[i] = x;
```
Terminology

- "Coherent" execution
  - Same instruction sequence applies to all elements operated upon simultaneously
  - Coherent execution is necessary for efficient use of SIMD processing resources
  - Coherent execution is NOT necessary for efficient parallelization across cores

- "Divergent" execution
  - A lack of coherence
More Complex Shader

```c
surface LGBrick ( float Ka = 1, Kd = 1; color brickcolor = color "rgb" (.6,.1,.1); color mortarcolor = color "rgb" (.6,.6,.6); float jagged = 0.006, brickvary = 0.3; float brickwidth = .25, brickheight = .08; float mortarthickness = .01; float rowvary = .25;) {
  float w, h, Nfactor;
  /* Determine how wide in s-t space one pixel projects to */
  float swidth = max (abs(Du(s)*du) + abs(Dv(s)*dv), MINFILTERWIDTH);
  float twidth = max (abs(Du(t)*du) + abs(Dv(t)*dv), MINFILTERWIDTH);

  normal Nf = faceforward (normalize(N),I);

  /* Make the shapes of the bricks vary just a bit */
  point PP2 = point noise (s/BMWIDTH, t/BMHEIGHT);
  float scoord = s + jagged * xcomp (PP2);
  float tcoord = t + jagged * ycomp (PP2);

  float ss = scoord / BMWIDTH;   /* Determine which brick the point is in */
  float tt = tcoord / BMHEIGHT;  /*                   "                   */
  swidth /= BMWIDTH;
  twidth /= BMHEIGHT;

  /* shift alternate rows */
  if (mod (tt*0.5, 1) > 0.5)
    ss += 0.5;

  float tbrick = floor (tt);   /* which brick row? */
  /* Shift the columns randomly by row */
  ss += rowvary * (noise (tbrick+0.5) - 0.5);

  float sbrick = floor (ss);   /* which brick column? */
  /* Now ss and tt are coords within the brick */
  ss -= sbrick;
  tt -= tbrick;

  /* Choose a color for the surface */
  if (swidth >= 1)
    w = 1 - 2*MWF;
  else w = clamp (boxstep(MWF-swidth,MWF,ss), max(1-MWF/swidth,0),1)
    - clamp (boxstep(1-MWF-swidth,1-MWF,ss), 0, 2*MWF/swidth);

  if (twidth >= 1)
    h = 1 - 2*MHF;
  else h = clamp (boxstep(MHF-twidth,MHF,tt), max(1-MHF/twidth,0),1)
    - clamp (boxstep(1-MHF-twidth,1-MHF,tt), 0, 2*MHF/twidth);

  /* Choose a brick color that varies from brick to brick */
  color bcolor = brickcolor * (1 + (brickvary * snoise (tbrick+(100*sbrick)+0.5)));
  color Ct = mix (mortarcolor, bcolor, w*h);

  Oi = Os;
  Ci = Os * Ct * (Ka * ambient() + Kd*diffuse(Nf));
}
```

http://renderman.org/RMR/Shaders/LGShaders/LGBrick.sl
More Application of SIMD in Reyes

- **Dicing:**
  - For each grid \((u,v)\) coordinate, evaluate surface geometry at that location

- **Bound**
  - Over all control points, find the min and max in the \(x, y,\) and \(z\) dimensions.

- **Dicing rates**
  - For each segment along iso-hulls, compute maximum distance between control points
Acknowledgements

- Slides 4-7 and 31-35 were provided by Kayvon Fatahalian.
Further References

- Solomon Boulos’s Syrah library on github
HW2 Overview: Morton Codes

- The diagram shows a 3D grid representation of Morton codes.
- Morton codes are a way of mapping multi-dimensional data to a one-dimensional space, typically used in spatial indexing and computer graphics.
- Each line represents a level of the Morton code, with 0 being the lowest level and 7 being the highest.
HW2 Overview: Morton Codes

```
    7   6   5   4   3   2   1   0
   ---------------
6 | .   .   .   .   .   .   .   . |
   ---------------
5 | .   .   .   .   .   .   .   . |
   ---------------
4 | .   .   .   .   .   .   .   . |
   ---------------
3 | .   .   .   .   .   .   .   . |
   ---------------
2 | .   .   .   .   .   .   .   . |
   ---------------
1 | .   .   .   .   .   .   .   . |
   ---------------
0 | .   .   .   .   .   .   .   . |
   ---------------
    0   1   2   3   4   5   6   7
```
HW2 Overview: Morton Codes

```
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
```
HW2 Overview: Morton Codes
HW2 Overview: Morton Codes
# HW2 Overview: Morton Codes

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>11</td>
<td>14</td>
</tr>
</tbody>
</table>
HW2 Overview: Morton Codes

<table>
<thead>
<tr>
<th>11</th>
<th>1010</th>
<th>1011</th>
<th>1110</th>
<th>1111</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1000</td>
<td>1001</td>
<td>1100</td>
<td>1101</td>
</tr>
<tr>
<td>01</td>
<td>0010</td>
<td>0011</td>
<td>0110</td>
<td>0111</td>
</tr>
<tr>
<td>00</td>
<td>0000</td>
<td>0001</td>
<td>0100</td>
<td>0101</td>
</tr>
</tbody>
</table>

00  01  10  11
Bit Interleaving

- - 9 9 9 8 8 8 7 7 7 6 6 6 5 5 5 4 4 4 3 3 3 2 2 2 1 1 1 0 0 0
Bit Interleaving
Bit Interleaving
Bit Interleaving
Bit Interleaving

```c
inline uint32_t Part1By2(float fx) {
    //
}

inline uint32_t EncodeMorton3(unsigned int v[3]) {
    return (Part1By2(v[2]) << 2) + (Part1By2(v[1]) << 1) + Part1By2(v[0]);
}
```

Part1By2

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -9 - 8 - 7 - 6 - 5 - 4 - 3 - 2 - 1 - 0
```

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -9 - - 8 - - 7 - - 6 - - 5 - - 4 - - 3 - - 2 - - 1 - - 0
```
## Straightforward Bit Interleaving

```c
inline uint32_t Part1By2(uint32_t x) {
    uint32_t result = 0;
    for (int i = 0; i < 10; ++i)
        result |= (x & (1 << i)) ? (1 << (2 * i)) : 0;
    return result;
}

inline uint32_t EncodeMorton3(unsigned int v[3]) {
    return (Part1By2(v[2]) << 2) + (Part1By2(v[1]) << 1) + Part1By2(v[0]);
}
```

---

**Part1By2**

```
- - - - - - - - - - - - - - - - - - - - - - 9 8 7 6 5 4 3 2 1 0
```

---

```
- - - - - - 9 - - 8 - - 7 - - 6 - 5 - - 4 - - 3 - - 2 - - 1 - - 0
```
Efficient Bit Interleaving

```c
inline uint32_t Part1By2(uint32_t x) {
    x &= 0b1111111111;
    // Bitwise operations
    x = (x ^ (x << 16)) & 0b111111110000000011111111;
    x = (x ^ (x <<  8)) & 0b00000011000000001111000000001111;
    x = (x ^ (x <<  4)) & 0b00000011000011000011000011000011;
    x = (x ^ (x <<  2)) & 0b00001001001001001001001001001001;
    return x;
}
```

[http://fgiesen.wordpress.com/2009/12/13/decoding-morton-codes/]