Systems and Languages for Graphics

Xuanda Yang

11/29
About Me

• $n$-th Year PhD student ($2 < n < 4$)
• Working with Prof. Tzu-Mao Li
• On systems and languages for graphics
  • (The topic today, what a coincidence)
• Fun fact: Play Dota 2 since 2013 (still bad)
• Photo description: happy me during summer
Overview

• Motivation: why we need systems & languages
• Unleash your GPU: From OpenGL to Brook
• Schedule your image processing kernels: Halide
• A modern modular shading approach: Slang
Systems & Languages

• Every field of CS is built with solid systems & languages

Deep Learning    Database    Web Development

• Why people invent them?
Systems & Languages

• Every field of CS is built with solid systems & languages
• Imagine writing a neural network in C/Fortran
  • What about stable diffusion in x86 assembly?
• Is the code correct?
• Is the code fast?
• How long does it take to implement it?
Systems & Languages

• Every field of CS is built with solid systems & languages

• Imagine writing a neural network in C/Fortran
  • What about stable diffusion in x86 assembly?

• Is the code correct?

• Is the code fast?

• How long does it take to implement it?

• **Correctness, Efficiency and productivity**
Graphics ❤ Systems & Languages

- Correctness, Efficiency and productivity
- Graphics values all of the three
  - Correctness: In-game interaction are correct and natural
  - Efficiency: Playing games in 240fps
  - Productivity: GTA VI can be made within this century
Graphics ❤ Systems & Languages

• Correctness, Efficiency and productivity

• Graphics has a lot of systems and programming languages
  • CUDA: GPU programming
  • OpenGL: Render your first triangle
  • Renderman shading language: Produce your favorite Pixar movie
  • .......

• This lecture only covers the tip of the iceberg
From SIGGRAPH 2023

• Largest annual conference of computer graphics

• What’s there?
From SIGGRAPH 2023

- Largest annual conference of computer graphics
- A collection of consumer GPUs all the way back to 1993
  - 1MB RAM, cannot run Minecraft
From SIGGRAPH 2023

- Largest annual conference of computer graphics
- A collection of consumer GPUs all the way back to 1993
  - 1MB RAM, cannot run Minecraft
- How do we program GPU in 2023?
- How did we program GPU in 2003?
- How do we implement ray tracers on GPU?
- How do we implement simulators on GPU?
OpenGL

• Recall HW3, it uses OpenGL to render a 3D bunny

```c
1 unsigned int VBO_vertex;
2 glGenBuffers(1, &VBO_vertex);
3 glBindBuffer(GL_ARRAY_BUFFER, VBO_vertex);
4 glBufferData(GL_ARRAY_BUFFER, ...);
5 glVertexAttribPointer(0 /* layout index */, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
6 glEnableVertexAttribArray(0);
7 unsigned int VBO_color;
8 glGenBuffers(1, &VBO_color);
9 glBindBuffer(GL_ARRAY_BUFFER, VBO_color);
10 glBufferData(GL_ARRAY_BUFFER, ...);
11 glVertexAttribPointer(1 /* layout index */, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
```

• Can we use this system to program a web application?
• What about a database?
• How should we program the shader?
OpenGL

- It was designed to do 3D rendering
- And almost only for 3D rendering
- Our first Domain-specific system
  - Only works for certain domain
  - Utilize domain knowledges when designing:
    - Parallel operations on triangles
    - Fast rasterization
Brook

- What if we want to do fancy stuff?
  - Physical simulation on GPU
- Turn simulation code into pixel shader and vertex shader : ((((
Brook

- A new programming language for data parallel computation
- Extend ANSI C
- Use streams as core computation model
- A more approachable programming model
Brook

• Example program

```c
1   kernel void k(float s<> , float3 f , float a[10][10] , out float o<>);
2
3   // define streams
4   float a<100>;
5   float b<100>;
6   float c<10, 10>;
7
8   streamRead(a, data1);
9   streamRead(b, data2);
10  streamRead(c, data3);
11
12  // Call kernel "k"
13  k (a, 3.2f, c, b);
14
15  streamWrite(b, result);
```

• That's it!

• No GLFooBar API calls

• No separate shader files
Brook Streams

- Streams
  - A data collection that operates similar computation
  - Has special read/write access
  - Disallow random access during computation
Brook

- Efficiency: High compute intensity programs fit perfectly
- Productivity: Very easy to learn for any C programmers
- Correctness: OpenGL API calls are carefully produced by the compiler
Brook Applications

ray-tracer

fft edge detect

segmentation

\[
\text{SAXPY} = \alpha \cdot \text{vector} + \text{vector}
\]

\[
\text{SGEMV} = \text{matrix} \cdot \text{vector}
\]

linear algebra
Brook

• Does it looks similar to something you know?

```c
1    kernel void k(float s<>, float3 f, float a[10][10], out float o<>);
2
3    // define streams
4    float a<100>;
5    float b<100>;
6    float c<10,10>;
7
8    streamRead(a, data1);
9    streamRead(b, data2);
10   streamRead(c, data3);
11
12   // Call kernel "k"
13   k (a, 3.2f, c, b);
14
15   streamWrite(b, result);
```
Brook: Beyond

- Does it look similar to something you know?

```c
kernel void k(float s<>, float f, float a[10][10], out float o<>); // define streams
float a<100>;
float b<100>;
float c<10,10>;
streamRead(a, data1);
streamRead(b, data2);
streamRead(c, data3);
// Call kernel "k"
k (a, 3.2f, c, b);
streamWrite(b, result);
```

Brook

```
Brook: CUDA
```
Image processing

- Consider a simple two-stage blur filter:

**Example and code adapted from Alex Reinking’s slides**
Image processing

• Consider a simple two-stage blur filter:

Blur by averaging pixel values in a 1x3 window

Example and code adapted from Alex Reinking’s slides
Consider a simple two-stage blur filter:

- Blur by averaging pixel values in a 3x1 window

Example and code adapted from Alex Reinking’s slides
Image<
uint8_t> blur(const Image<
uint8_t> &in) {
    // handle boundary
    Image<
uint16_t> in16(in.width() + 2, in.height() + 2);
    for (int x = 0; x < in16.width(); ++x)
        for (int y = 0; y < in16.height(); ++y)
            in16(x, y) = in(clamp(x-1,0, in.width()-1), clamp(y-1,0, in.height()-1));
    // horizontal blur
    Image<
uint16_t> blur_x(in16.width()-2, in16.height());
    for (int x = 1; x <= blur_x.width(); ++x)
        for (int y = 0; y < blur_x.height(); ++y)
            blur_x(x-1, y) = (in16(x-1,y) + in16(x,y) + in16(x+1,y))/3;
    // vertical blur
    Image<
uint8_t> blur_y(blur_x.width(), blur_x.height()-2);
    for (int x = 0; x < blur_y.width(); ++x)
        for (int y = 1; y <= blur_y.height(); ++y)
            blur_y(x, y-1) = (blur_x(x,y-1) + blur_x(x,y) + blur_x(x,y+1))/3;
    return blur_y;
}
```cpp
Image<uint8_t> blur(const Image<uint8_t> &in) {
    // handle boundary
    Image<uint16_t> in16(in.width() + 2, in.height() + 2);
    for (int x = 0; x < in16.width(); ++x)
        for (int y = 0; y < in16.height(); ++y)
            in16(x, y) = in(clamp(x-1,0,in.width()-1),clamp(y-1,0,in.height()-1));
    // horizontal blur
    Image<uint16_t> blur_x(in16.width()-2, in16.height());
    for (int x = 1; x <= blur_x.width(); ++x)
        for (int y = 0; y < blur_x.height(); ++y)
            blur_x(x-1, y) = (in16(x-1,y) + in16(x,y) + in16(x+1,y))/3;
    // vertical blur
    Image<uint8_t> blur_y(blur_x.width(), blur_x.height()-2);
    for (int x = 0; x < blur_y.width(); ++x)
        for (int y = 1; y <= blur_y.height(); ++y)
            blur_y(x, y-1) = (blur_x(x,y-1) + blur_x(x,y) + blur_x(x,y+1))/3;
    return blur_y;
}

Bad cache locality -> bad performance
(100ms on a 4-megapixel image, intel i9-9700x 32GB ram)

Example and code adapted from Alex Reinking’s slides
```cpp
Image<
  uint8_t
>
blur(const Image<
  uint8_t
> &in) {
  // handle boundary
  Image<
    uint16_t
  > in16(in.width() + 2, in.height() + 2);
  for (int x = 0; x < in16.width(); ++x)
    for (int y = 0; y < in16.height(); ++y)
      in16(x, y) = in(clamp(x-1, 0, in.width()-1),
                      clamp(y-1, 0, in.height()-1));
  // horizontal blur
  Image<
    uint16_t
  > blur_x(in16.width()-2, in16.height());
  for (int y = 0; y < blur_x.height(); ++y)
    for (int x = 1; x <= blur_x.width(); ++x)
      blur_x(x-1, y) = (in16(x-1,y) + in16(x,y) + in16(x+1,y))/3;
  // vertical blur
  Image<
    uint8_t
  > blur_y(blur_x.width(), blur_x.height()-2);
  for (int x = 0; x < blur_y.width(); ++x)
    for (int y = 1; y <= blur_y.height(); ++y)
      blur_y(x, y-1) = (blur_x(x,y-1) + blur_x(x,y) + blur_x(x,y+1))/3;
  return blur_y;
}
```

Fix it by swapping the order (8ms/13x speedup)
void box_filter_3x3(const Image &in, Image &blury) {
__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 blurx[(256/8)*(32+2)]; // allocate tile blurx array
for (int xTile = 0; xTile < in.width(); xTile += 256) {
__m128i *blurxPtr = blurx;
for (int y = -1; y < 32+1; y++) {
const uint16_t *inPtr = &(in[yTile+y][xTile]);
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(blurxPtr++, avg);
inPtr += 8; }
blurxPtr = blurx;
}
for (int y = 0; y < 32; y++) {
__m128i *outPtr = (__m128i *)&(blury[yTile+y][xTile]);
for (int x = 0; x < 256; x += 8) {
a = _mm_load_sil128(blurxPtr+(2*256)/8);
b = _mm_load_sil128(blurxPtr+256/8);
c = _mm_load_sil128(blurxPtr++);
avg = _mm_mulhi_epi16(sum, one_third);
sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
__mm_store_sil128(outPtr++, avg); }})}}
1 void box_filter_3x3(const Image &in, Image &blur) {
2   __m128i one_third = _mm_set1_epi16(21846);
3   #pragma omp parallel for
4   for (int yTile = 0; yTile < in.height(); yTile += 32) {
5      __m128i a, b, c, Sum, Avg;
6      __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
7      for (int xTile = 0; xTile < in.width(); xTile += 256) {
8         __m128i *blurxPtr = blurx;
9         for (int y = -1; y < 32+1; y++) {
10            const uint16_t *inPtr = &(in[yTile+y][xTile]);
11            for (int x = 0; x < 256; x += 8) {
12               a = __mm_loadu_si128((__m128i*)(inPtr-1));
13               b = __mm_loadu_si128((__m128i*)(inPtr+1));
14               c = __mm_load_si128((__m128i*)(inPtr));
15               sum = __mm_add_epi16(__mm_add_epi16(a, b), c);
16               avg = __mm_mulhi_epi16(sum, one_third);
17               __mm_store_si128(blurxPtr++, avg);
18               inPtr += 8; }}}
19      blurxPtr = blurx;
20      for (int y = 0; y < 32; y++) {
21         __m128i *outPtr = (__m128i*)(&blur[yTile+y][xTile]);
22         for (int x = 0; x < 256; x += 8) {
23            a = __mm_load_si128(blurxPtr+(2*256)/8);
24            b = __mm_load_si128(blurxPtr+256/8);
25            c = __mm_load_si128(blurxPtr++);
26            avg = __mm_mulhi_epi16(sum, one_third);
27            sum = __mm_add_epi16(__mm_add_epi16(a, b), c);
28            __mm_store_si128(outPtr++, avg); }}
29   }}

void box_filter_3x3(const Image &in, Image &blury) {
    __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 blurx[(256/8)*(32+2)]; // allocate tile blurx array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurxPtr = blurx;
            for (int y = -1; y < 32+1; y++) {
                const uint16_t *inPtr = &(in[yTile+y][xTile]);
                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(blurxPtr++, avg);
                    inPtr += 8; }
            }
            blurxPtr = blurx;
            for (int y = 0; y < 32; y++) {
                __m128i *outPtr = (__m128i *)&(blury[yTile+y][xTile]);
                for (int x = 0; x < 256; x += 8) {
                    a = __mm_load_si128(blurxPtr+((x+256)/8));
                    b = __mm_load_si128(blurxPtr+256/8);
                    c = __mm_load_si128(blurxPtr++);
                    avg = __mm_mulhi_epi16(sum, one_third);
                    sum = __mm_add_epi16(__mm_add_epi16(a, b), c);
                    __mm_store_si128(outPtr++, avg); }
            }
        }
    }
}
void box_filter_3x3(const Image &in, Image &blury) {
    __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 blurx[(256/8)*(32+2)]; // allocate tile blurx array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurxPtr = blurx;
            for (int y = -1; y < 32+1; y++) {
                const uint16_t *inPtr = &(in[yTile+y][xTile]);
                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(blurxPtr++, avg);
                    inPtr += 8; }}
            blurxPtr = blurx;
            for (int y = 0; y < 32; y++) {
                __m128i *outPtr = (__m128i *)&(blury[yTile+y][xTile]);
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_load_si128(blurxPtr+(2*256)/8);
                    b = _mm_load_si128(blurxPtr+256/8);
                    c = _mm_load_si128(blurxPtr++);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    _mm_store_si128(outPtr++, avg); }}
            }}
        }
    }
}
Halide

• Decouple Algorithms and Schedules

• Algorithm: functional definition of computation
  • Horizontal blur: \( f(x) = \frac{g(x - 1) + g(x) + g(x + 1)}{3} \)

• Schedule: optimization choices
  • Loop Fusion: combining multiple loops together
  • Vectorization: using multi-bit instructions like VPMADDUBSW
  • Different backends: GPU, CPU, Custom Hardware…
Algorithm

```plaintext
1 Func blur(Func input) {
2   Var x, y;
3   Func input_16, blur_x, blur_y, result;
4   // handle boundary
5   Func clamped = BoundaryConditions::repeat_edge(input);
6   input_16(x, y) = cast<uint16_t>(clamped(x, y));
7   // horizontal blur
8   blur_x(x, y) = (input_16(x - 1, y) + input_16(x, y) + input_16(x + 1, y)) / 3;
9   // vertical blur
10  blur_y(x, y) = (blur_x(x, y - 1) + blur_x(x, y) + blur_x(x, y + 1)) / 3;
11  result(x, y) = cast<uint8_t>(blur_y(x, y));
12  return result;
13 }
```

Example and code adapted from Alex Reinking’s slides
Schedules

• `compute_root` computes everything and store them in a big array before reading from it

• `vectorize` vectorizes loop for a given width

Example and code adapted from Alex Reinking’s slides
Schedules

- **compute_at** interleaves the computation of two funcs by moving the loops of producing loop into the ones of consuming func

- **tile** chops up the loops into given tile shape

```
1 result.compute_root()
2     .tile(x, y, xi, yi, 128, 24)
3     .vectorize(x, 32)
4
5 blur_x.compute_at(result, yi)
6     .vectorize(x, 32);
7
8 input_16.compute_at(result, x)
9     .vectorize(x, 32);
```

1.2ms

Example and code adapted from Alex Reinking’s slides
Halide

- Decoupling implies tons of benefits:
  - **Correctness**: different schedules should produce the same output
  - **Productivity**: high-level abstraction of different optimization choices
  - Halide compiler, plus a more productivity-unleashed programmer
  - **Efficiency**: several lines of schedule code usually perform better than hundreds of CUDA/C++ code
Halide Applications

Adobe Photoshop  Pixel Camera  Instagram
Complicated Shading System

- Recall again our OpenGL programming experience in HW3
- If we need to implement a material systems with 20+ materials
- In separate files: diffuse.cpp, metal.cpp, ....
- Now how do we extend our OpenGL shader to adapt all of them?
Complicated Shading System

Figure from Yong He's slides
Complicated Shading System

- Implement a material systems with 20+ materials
- C++ object hierarchies
  - Dynamic dispatch, slow
- Use the preprocessor
  - Simple text replacement, error prone
  - Hard to maintain

Examples are from Yong He’s slides
Slang: A Modular Shading Approach

• A new shading language aim to replace GLSL/HLSL

• Extend HLSL with programming constructs that make systems more:
  • Modular
  • Composable
  • Extensible
  • Performant
Slang

- Slang provides first-class interface to constrain class hierarchies
- Correctness: Static type checking rather than text replacement

```csharp
// Interface ILight { float3 computeLighting(); }

struct SpotLight : ILight { float3 computeLighting() { // ... } }

struct SkyLight : ILight { float3 computeLighting() { // ... } }

void forwardPassShader<TGeometry: IGeometry,
TLight: ILight,
TMaterial: IMaterial>(
TGeometry geometry,
TLight light,
TMaterial material) {
    // ...
}
```

Examples are from Yong He's slides
Slang

- Slang provides specialization via generics
- Efficiency: Similar to C++ templating, no more runtime dispatch

```c

t void forwardPassShader<TGeometry: IGeometry,
   TLight: ILight,
   TMaterial: IMaterial>(
   TGeometry geometry,
   TLight light,
   TMaterial material) {
   // ...
}

struct DisplacementGeometry: IGeometry {
   // ...
};

struct SkyLight: ILight {
   // ...
};

struct Metal: IMaterial {
   // ...
};

forwardPassShader<DisplacementGeometry, SkyLight, Metal>
```

Examples are from Yong He’s slides
Slang

- Slang powers the NVIDIA Falcor renderer
  
  - [https://github.com/NVIDIAGameWorks/Falcor](https://github.com/NVIDIAGameWorks/Falcor)

- Slang recently becomes differentiable!
Summary

- Graphics needs solid programming systems & languages supports
- Domain-specific languages utilize domain-knowledge in Graphics that
  - Making code easier to write, boost productivity
  - Stronger checks and abstractions makes your code more correct
  - Powerful compiler generates performant programs