Python on the GPU:
Implications for Many-core

Nicholas C. Bray (nbray1@uiuc.edu)
Acknowledgments

- Ralph Johnson
- UPCRC
What's the Problem?

- Hardware is becoming harder to program
  - CPU programming model unsustainable
  - Parallelism is **not** the problem
- Software is becoming larger
  - We need less code and clearer code
- How can we resolve this tension?
Real-time Rendering Challenges

- Coprocessor programming
  - Limited language
  - Few abstractions
- Two code bases
  - Functions
  - Data Structures
  - Semantics
- Glue Code
  - Serialization
  - Relationships between shaders
- Heterogeneous multi-core? Stream programming?
- Inherently Complex
  - Many algorithms working together
  - Algorithms span CPU and GPU
High-level Programming

- Programmers will create abstractions as needed
  - Code generation for shaders
  - Hard to optimize
- Python
  - 7th most popular language
  - Designed for rapid development
  - 100x-1000x slower, yet widely used!
  - Static compilation is an open problem
    - Starkiller, PyPy, Shed Skin
The Big Idea

- Python shaders on the GPU
  - High-level shader programming
  - Unify CPU + GPU code base
  - No glue code
- Python + GPU: a good idea
  - PyGPU
  - Copperhead
What is PyStream?

- Python Shader
- PyStream Compiler
- GLSL

- Python
- 30k SLOC
- - Restrictions
- + Shader Format
- GPU
- CPU
PyStream Language Restrictions

- At least as expressive as GLSL
- No recursion
  - Function calls
  - Data structures
- Closed world
- Numeric types
- Engineering restrictions
  - Exceptions, keyword args, closures, generators
- Some data structures cannot be emulated
- Most restrictions are applied after compilation
class **FullScreenEffect**(object):

    def **shadeVertex**(self, context, pos, texCoord):
        context.position = pos
        return texCoord,

    def **shadeFragment**(self, context, texCoord):
        context.colors = (self.process(texCoord),)

class **AmbientPass**(FullScreenEffect):

    def **process**(self, texCoord):
        # Sample the underlying geometry
        g = self.gbuffer.sample(texCoord)
        # Sample the ambient occlusion
        ao = self.ao.texture(texCoord).xyz
        # Calculate the lighting
        ambientLight = self.env.ambientColor(g.normal)*ao
        # Modulate the output
        return vec4(g.diffuse*ambientLight, 1.0)
# Creating the shader
shader = AmbientPass()
shader.gbuffer = gbuffer
shader.env = env
shader.ao = ao

# Using the shader to draw data
draw(shader, posBuffer, texBuffer)
class CompiledAmbientPass(pystreamruntime.BaseCompiledShader):
    def _bindUniforms(self, shader):
        bogus = tests.full.physics.AmbientPass.ao.__get__(shader)
        self.bind_uniform_sampler2D('samplerGroup2', bogus)
        bogus_0 = tests.full.physics.AmbientPass.gbuffer.__get__(shader)
        bogus_1 = tests.full.physics.GBuffer.surface.__get__(bogus_0)
        self.bind_uniform_sampler2D('samplerGroup0', bogus_1)
        bogus_2 = tests.full.physics.GBuffer.position.__get__(bogus_0)
        self.bind_uniform_sampler2D('samplerGroup1', bogus_2)
        bogus_3 = tests.full.physics.GBuffer.normal.__get__(bogus_0)
        self.bind_uniform_sampler2D('samplerGroup3', bogus_3)
        bogus_4 = tests.full.physics.AmbientPass.env.__get__(shader)
        bogus_5 = tests.full.physics.Environment.cameraToEnvironment.__get__(bogus_4)
        self.bind_uniform_mat4('uni_cameraToEnvironment_mat4', bogus_5)
        self.bind_uniform_samplerCube('samplerGroup4', bogus_6)

    def bindStreams(self, pos, texCoord):
        self.bind_stream_vec4('inp_io_vec4', pos)
        self.bind_stream_vec2('inp_io_vec2', texCoord)

class AmbientPass(FullScreenEffect):
    def process(self, texCoord):
        g = self.gbuffer.sample(texCoord)
        ao = self.ao.texture(texCoord).xyz
        ambientLight = self.env.ambientColor(g.normal)*ao
        return vec4(g.diffuse*ambientLight, 1.0)

uniform sampler2D samplerGroup0;
uniform sampler2D samplerGroup1;
uniform sampler2D samplerGroup2;
uniform sampler2D samplerGroup3;
uniform samplerCube samplerGroup4;
layout(shared) uniform uni {
    uniform mat4 uni_cameraToEnvironment_mat4;
};
in vec4 inp_io_vec4;
in vec2 inp_io_vec2;
out vec4 out_io_vec4;
void main()
{
    gl_Position = inp_io_vec4;
    inp_io_vec2_1 = inp_io_vec2;
}

out vec2 inp_io_vec2_1;
in vec4 inp_io_vec4;
in vec2 inp_io_vec2;
void main()
{
    gl_Position = inp_io_vec4;
    inp_io_vec2_1 = inp_io_vec2;
}

uniform sampler2D samplerGroup0;
uniform sampler2D samplerGroup1;
uniform sampler2D samplerGroup2;
uniform sampler2D samplerGroup3;
uniform samplerCube samplerGroup4;
layout(shared) uniform uni {
    uniform mat4 uni_cameraToEnvironment_mat4;
};
in vec2 inp_io_vec2_1;
in vec4 out_io_vec4;
out vec4 out_io_vec4;
void main()
{
    vec3 albedo, normal, cameraNormal, ambient, ao;
    albedo = texture(samplerGroup0, inp_io_vec2_1).xyz;
    normal = normalize(texture(samplerGroup1, inp_io_vec2_1).xyz);
    cameraNormal = uni_cameraToEnvironment_mat4*vec4(normal, 0.0).xyz;
    ambient = texture(samplerGroup4, cameraNormal);
    ao = texture(samplerGroup2, inp_io_vec2_1).xyz;
    out_io_vec4 = vec4(diffuse*ambient*ao, 1.0);
}
100% Python
- No glue code
- 147,000x faster
- No overhead in generated shaders

Deferred Rendering

Image Quality
- Gamma-correct HDR
- SSAO
- Parallax Occlusion Mapping

~60 fps
## Example Shader Programs

<table>
<thead>
<tr>
<th>Shader</th>
<th>Pixels</th>
<th>GPU</th>
<th>GPU/Pixels</th>
<th>CPU</th>
<th>CPU/Pixels</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>material</td>
<td>0.77</td>
<td>4.33 ms</td>
<td>5.62 ms</td>
<td>169.8 s</td>
<td>220.5 s</td>
<td>39,211x</td>
</tr>
<tr>
<td>skybox</td>
<td>0.27</td>
<td>0.22 ms</td>
<td>0.81 ms</td>
<td>9.6 s</td>
<td>35.5 s</td>
<td>43,568x</td>
</tr>
<tr>
<td>ssao</td>
<td>1.00</td>
<td>1.44 ms</td>
<td>1.44 ms</td>
<td>444.9 s</td>
<td>444.9 s</td>
<td>308,958x</td>
</tr>
<tr>
<td>bilateral</td>
<td>2.00</td>
<td>2.97 ms</td>
<td>1.49 ms</td>
<td>858.2 s</td>
<td>429.1 s</td>
<td>288,956x</td>
</tr>
<tr>
<td>ambient</td>
<td>1.00</td>
<td>0.84 ms</td>
<td>0.84 ms</td>
<td>64.1 s</td>
<td>64.1 s</td>
<td>76,310x</td>
</tr>
<tr>
<td>light</td>
<td>5.00</td>
<td>4.75 ms</td>
<td>0.95 ms</td>
<td>635.5 s</td>
<td>127.1 s</td>
<td>133,789x</td>
</tr>
<tr>
<td>blur</td>
<td>4.00</td>
<td>2.14 ms</td>
<td>0.54 ms</td>
<td>296.8 s</td>
<td>74.2 s</td>
<td>138,692x</td>
</tr>
<tr>
<td>post</td>
<td>0.23</td>
<td>2.20 ms</td>
<td>9.57 ms</td>
<td>101.8 s</td>
<td>442.6 s</td>
<td>46,272x</td>
</tr>
<tr>
<td>total</td>
<td>14.27</td>
<td>17.59 ms</td>
<td>1.23 ms</td>
<td>2580.7 s</td>
<td>180.8 s</td>
<td>146,712x</td>
</tr>
</tbody>
</table>

AMD Athlon 64 X2 3800+ CPU - NVidia 9800 GT GPU – Windows XP - CPython 2.6.4 – 1024x1024 resolution
High-level Optimization

![Graph showing FPS vs. Objects for OpenGL3 and OpenGL2](image-url)
Challenges Compiling Python

- Kitchen sink design
- Complicated semantics
  - What is an attribute? (field, ad hoc, method, etc.)
  - What is addition?
  - Operations can be customized
- No initial information
  - No type information or call graph
  - Highly polymorphic
- “Declarations” are mutable structures
  - Must disallow dynamic code generation
- Critical information passes through the heap
PyStream Compiler Pipeline

- Pointer Analysis
- Method Fusion
- Function Cloning
- Arg. Normalization
- Exhaustive Inlining
- Reanalysis + Load/Store
- Tree Transform
- Output Flattening
- Field Transform
- Translate to GLSL

- Infer Information
- Remove Abstraction Overhead
- Eliminate Memory Operations
- Bridge Semantic Gap
What was used
- Pointer analysis
- Direct references
- Constant folding
- Dead code elimination
- Function cloning
- Function inlining
- Load / store elimination
- Python-specific transformations
  - Method fusion and argument normalization
- Shader-specific transformations
  - Tree transform, output flattening and field transform
Engineering PyStream

- What did not work
  - Type inference
  - Classic heap sensitivity
  - BDD analysis algorithms
  - Tracked cell analysis
  - Dataflow IR
- What was not even tried
  - Loop analysis
High-level Analysis

- Used by Starkiller and PyPy
- Attributes are fields
  - … except if they're methods
  - … except if they're overridden
  - Ignore the other cases
- Operations are resolved by the analysis
Low-level Analysis

- Interpreter and run time → analyzed code
  - Interpreter becomes part of the program
  - Add direct calls, memory operations
  - Keep “loops” in the analysis
- Operations → calls
- Model memory at the interpreter level
  - Field vs. method?
- Analysis and optimization are much simpler
  - Looks like C + indirections
  - Should scale well
- Interpreter vs. JIT vs. pointer analysis
def interpreter_add(self, other):
    result = NotImplemented

    # Forward add
    selfcls <load>(self, 'LowLevel', 'type')
    clsDict = <load>(selfcls, 'LowLevel', 'dictionary')
    if <check>(clsDict, 'Dictionary', '__add__'):
        meth = <load>(clsDict, 'Dictionary', '__add__')
        result = meth(self, other)

    if result is NotImplemented:
        # Reverse add
        othercls = <load>(other, 'LowLevel', 'type')
        clsDict = <load>(othercls, 'LowLevel', 'dictionary')
        if <check>(clsDict, 'Dictionary', '__radd__'):
            meth = <load>(clsDict, 'Dictionary', '__radd__')
            result = meth(other, self)

    if result is NotImplemented:
        # Invalid operation
        raise ...

    return result

def simple(a, b):
    return a+b
def process(self, texCoord):
    g = self.gbuffer.sample(texCoord)
    ao = self.ao.texture(texCoord).xyz
    ambientLight = self.env.ambientColor(g.normal)*ao
    return vec4(g.diffuse*ambientLight, 1.0)
Method Fusion

- Fuse getattr + call → method call
  - Requires interprocedural analysis
- High-level information is great!
- High-level information is awful!
- Compiler vs. programmer
- Reconstruct when useful

\[
m = o.f \\
m(a, b) \rightarrow <\text{method } o \ f>(a, b)
\]
Python to GLSL

- Simplified by prior optimization
- Copy values wherever possible
- No memory allocation: bound structure size
  - No recursive data structures
  - No variable-size container objects
  - Dictionary lookups are difficult
- Volatility
  - Object modified while held by 2+ references
  - Cannot be soundly copied
  - Does not occur in the example shaders
Programmability and performance can co-exist
- Existing techniques + R&D
- Work at a lower level when necessary
- No cookbooks, yet

Overcoming language/architecture mismatch
- Easy, 90% of the time?
- New techniques needed for the rest
  - Change our expectations?
- Program at an even higher level
Backup sides start here
Object-oriented Polymorphism
What objects can a variable point to?
- What is the type of a variable?
- What is the call graph?
- What memory operations are dependent? *

Low-level analysis simplifies the problem
- Almost C, but not quite...
- Large amounts of polymorphism
Parametric polymorphism: CPA contexts
- Implicit “template” functions
Data polymorphism: extended types
- Bound method objects
Ad hoc polymorphism: control sensitivity
- No types, no function overloading
- Python idiom: explicit decoding of types

def add(a, b):
    return a+b

print add(1, 2)
print add('hello', '!!')

class vec2(object):
    def __init__(self, x, y=None):
        if isinstance(x, vec2):
            assert y is None
            self.x = x.x
            self.y = x.y
        else:
            self.x = x
            self.y = y
temp = <interpreterLoadGlobal>(internal_self, 'vec4')
return temp(vec3data, 1.0)

derived = <class 'shader.vec.vec4'>
return derived(vec3data, 1.0)

return <type_call, <class 'shader.vec.vec4'>>(vec3data, 1.0)
Novel Transform: Methods & Args

\[ m = o.f \]
\[ m(a, b) \]

\[ <method o f>(a, b) \]

```python
def type_call(cls, *vparam):
    obj = cls.__new__(cls, *vparam)
    obj.__init__(*vparam)
    return obj
```

```python
def type_call(cls, p0, p1):
    obj = cls.__new__(cls, p0, p1)
    obj.__init__(p0, p1)
    return obj
```
Ambient Lighting Shader
@property
def xyz(self):
    return vec3(self.x, self.y, self.z)
Ambient Lighting - Optimized
Ambient Lighting - Inlined
Eliminating Memory Operations

- Memory operations may remain
  - Reading input structures
  - Building output structures
  - Ambiguous memory operations
  - Allocated in loops, carried through merges
- Immutability → aliasing does not matter
  - Assume inputs are tree shaped
  - Coerce outputs into tree shape
- Transform fields into local variables
  - Unique and mutually exclusive
- Some memory operations can survive
Must emulate references in GLSL

- Emulate reference semantics
  - Inline fields into references
    - No recursive data structures!
  - Inline intrinsic objects into references
  - Copy values between references when possible
- Ambiguous memory operations $\rightarrow$ array
Volatility

- Values can be indirectly modified
  - Held by more than one reference and modified
- Store in “pools” and hold by index
- Rare in practice?
  - Can ignore side effects of GLSL functions
  - Few memory operations → few side effects
  - SSA + load elimination → uniquely held
- Coding style: allocate rather than modify

<table>
<thead>
<tr>
<th></th>
<th>Multiple Refs</th>
<th>Uniquely Held</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutable</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Immutable</td>
<td>45</td>
<td>307</td>
</tr>
<tr>
<td>Samplers</td>
<td>0</td>
<td>22</td>
</tr>
</tbody>
</table>
What Can Not Be Emulated?

- Data structures of unbounded size
  - Recursive data structures
  - Container objects, most of the time
  - Long integers and strings
  - Resource limits
- Unbounded numbers of volatile values
- Dictionary lookups
  - … unless keys can be explicitly enumerated
- These cases can be compiled away