Fabric Engine and KL
LLVM for 3D Digital Content Creation
The Plan

- Overview of Fabric Engine
- Uses of LLVM within KL
- Looking forward
What is Fabric Engine?

- Digital Content Creation (DCC) framework
- Standalone applications built on PySide
What is Fabric Engine?

• Digital Content Creation (DCC) framework
• Standalone application built on PySide
• Integration with existing DCC tools (Splice)
What is Fabric Engine?

- Digital Content Creation (DCC) framework
- Standalone application built on PySide
- Integration with existing DCC tools (Splice)
- In use by production studios
Why does Fabric Engine exist?

- Many DCC tools are old
- Closed and not configurable
- Writing plugins is hard (C++) or slow (Python)
- Development held back by software limitations
Fabric Engine

- KL language + dependency graph
- Host language bindings (Python/C++)
- Simple multithreading (MapReduce/PEX)
- Fully cross-platform (Windows/Linux/OSX)
- Code portable among other DCC applications
How does KL help?

- Ease of Python with performance of threaded C++
- Write once, use anywhere
- Crash–free and updateable on the fly
- Supports extensions for integration with existing libraries
- Target selection at runtime (CPU or GPU)
require Math;

operator parallel_init::<index>>(io Vec3 v[], Float32 m) {
  v[index] = Vec3(index*m, index, index*2);
}

function initialize(io Vec3 v[]) {
  parallel_init<::v.size()>(v, 3.3);
}

operator entry() {
  Vec3 v[];
  v.resize(1024);
  initialize(v);
  report(v);
}
So how does KL achieve this?

- In short: LLVM!
- MCJIT–backed
- Fabric Core compiler + scheduler
- Let's look at some specifics...
What's important for KL?

- Ease of use
- Fastest possible execution time
- Minimal memory footprint
- No significant startup delay
KL – Compilation Passes

- JIT languages slower to start than interpreted (ex. Python)
- Want maximum performance from LLVM
- Two-pass compilation
  - First unoptimized compilation pass
  - Fully optimized code generated in background
KL – Compilation Passes

- Sample case: CityAtNight.py
- 37k lines KL
- = 1.8M lines IR (pre-opt)

<table>
<thead>
<tr>
<th>Method</th>
<th>Startup time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upfront optimization</td>
<td>2m56s</td>
</tr>
<tr>
<td>Background optimization</td>
<td>0m37s</td>
</tr>
</tbody>
</table>
KL – Caching

- Using MCJIT ObjectCache since its introduction
- Cache both IR and objects
- Key based on hash of KL AST
- Use of IR “stubs” with cached data
function initialize(io Vec3 v[]) {
  parallel_init <<<v.size()>>>(v, 3.3);
}

define private void @function.initialize.io_A50.STVec3_VA(%STVec3_VA.Bits** nocapture %v) {
  entry:
  %cpuArgsStructPtr = alloca %internal.parallel_init.stub.cpu.args.00
  %cpuArgsStructHostPtr = alloca %internal.parallel_init.stub.cpu.args
  %0 = call i32 @method.size.in_A50.STVec3_VA(%STVec3_VA.Bits** %v)
  %cpuArgValuePtr.0 = getelementptr inbounds %internal.parallel_init.stub.cpu.args.00* %cpuArgsStructPtr, i32 0,
  store %STVec3_VA.Bits** %v, %STVec3_VA.Bits*** %cpuArgValuePtr.0
  %cpuArgValuePtr.1 = getelementptr inbounds %internal.parallel_init.stub.cpu.args.00* %cpuArgsStructPtr, i32 0,
  store float 0x400A666660000000, float* %cpuArgValuePtr.1
  %cpuArgsStructVoidPtr = bitcast %internal.parallel_init.stub.cpu.args.00* %cpuArgsStructPtr to i8*
  %nonZeroCountCond = icmp ne i32 0, 0
  br i1 %nonZeroCountCond, label %ep.nonZeroCount, label %ep.done

define private void @function.initialize.io_A50.STVec3_VA(%STVec3_VA.Bits** nocapture %v) {
  entry:
  unreachable
}
KL – Caching

- Sample case: CityAtNight.py
- 37k lines KL
- = 1.8M lines IR (pre-opt)

<table>
<thead>
<tr>
<th>Method</th>
<th>Startup time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upfront optimization</td>
<td>2m56s</td>
</tr>
<tr>
<td>Background optimization</td>
<td>0m37s</td>
</tr>
<tr>
<td>From cache</td>
<td>0m4s</td>
</tr>
</tbody>
</table>
KL – Linking

- Extensions export functions and methods
- Core links swappable function pointer into KL
- Same mechanism used in optimization pass
- Allows updating linked runtime code
- Explicit 'inline' modifier for extension functions
// equals operator
inline Boolean == (Vec3 a, Vec3 b) {
  return a.x == b.x && a.y == b.y && a.z == b.z;
}

// not equals operator
inline Boolean != (Vec3 a, Vec3 b) {
  return a.x != b.x || a.y != b.y || a.z != b.z;
}

// adds to vectors
inline Vec3 + (Vec3 a, Vec3 b) {
  return vecAdd(a, b);
}

// adds a vector to this one
inline Vec3 += (Vec3 other) {
  this = this + other;
}

// subtracts two vectors
inline Vec3 - (Vec3 a, Vec3 b) {
  return vecSub(a, b);
}

// subtracts a vector from this one
## KL – Linking

- Sample case: SPHSimulation.py

<table>
<thead>
<tr>
<th>Method</th>
<th>Startup time</th>
<th>FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inline everything</td>
<td>2m11s</td>
<td>26</td>
</tr>
<tr>
<td>Nothing inlined</td>
<td>0m34s</td>
<td>22</td>
</tr>
<tr>
<td>Selective use of 'inline'</td>
<td>0m35s</td>
<td>26</td>
</tr>
</tbody>
</table>
KL – Memory Use

- After compilation want minimal memory use
- LLVM 3.4: delete Module after compile
- Still need multiple ExecutionEngines
**KL – Memory Use**

- Sample case: Crowd.py
- With ObjectCache

<table>
<thead>
<tr>
<th>Method</th>
<th>RSS (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full IR + no removeModule</td>
<td>797</td>
</tr>
<tr>
<td>Stub IR + no removeModule</td>
<td>428</td>
</tr>
<tr>
<td>Full IR + removeModule</td>
<td>367</td>
</tr>
<tr>
<td>Stub IR + removeModule</td>
<td>356</td>
</tr>
<tr>
<td>Shared ExecutionEngine</td>
<td>296</td>
</tr>
</tbody>
</table>
KL – GPU Compute

- KL code run without modification on CPU or GPU
- AMD HSA hardware shared memory
- Nvidia Cuda 6 “shared memory” via driver
- Speedup varies by application and hardware but up to 10x faster
- First release coming in May 2014
KL – GPU Compute

- Sample case: Mandelbrot.py
- Standard desktop hardware

<table>
<thead>
<tr>
<th>Target</th>
<th>FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core i7–3770k @ 3.50GHz</td>
<td>3.7</td>
</tr>
<tr>
<td>NVIDIA Quadro K5000</td>
<td>23.5</td>
</tr>
</tbody>
</table>
operator computePixels(
    Size cols,
    Size rows,
    io Image2DRGB image,
    Complex64 center,
    Float64 zoom,
    Size maxIterations,
    Boolean useGPU
) {
    image.resize(cols, rows);
    computePixel<<<image.pixels.size()@useGPU>>>(
        cols,
        rows,
        image.pixels,
        center,
        zoom,
        maxIterations
    );
    image.incrementVersion();
}
KL – Debugging

- Dwarf info via LLVM DIBuilder
- LLDB JIT support
- Breakpoints, threads, variable inspection, etc.
- Python + PySide LLDB front-end
<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Type</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>pos</td>
<td>0x0000000000005e00</td>
<td>Vec3</td>
<td>-</td>
</tr>
<tr>
<td>z</td>
<td>0x0000000000005e00</td>
<td>Ouat</td>
<td>-</td>
</tr>
<tr>
<td>segment</td>
<td>0x000000000000a148</td>
<td>CarTrajectorySegment*</td>
<td>-</td>
</tr>
<tr>
<td>type</td>
<td>0</td>
<td>int</td>
<td>-</td>
</tr>
<tr>
<td>startPt</td>
<td>428.725006</td>
<td>float</td>
<td>-</td>
</tr>
<tr>
<td>z</td>
<td>-206.649994</td>
<td>float</td>
<td>-</td>
</tr>
<tr>
<td>endPt</td>
<td>8</td>
<td>float</td>
<td>-</td>
</tr>
<tr>
<td>endSpeed</td>
<td>4</td>
<td>float</td>
<td>-</td>
</tr>
<tr>
<td>timeLapse</td>
<td>2</td>
<td>float</td>
<td>-</td>
</tr>
<tr>
<td>currSpeed</td>
<td>5.10337448</td>
<td>float</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>9.48889255</td>
<td>float</td>
<td>-</td>
</tr>
</tbody>
</table>

```
lightSourceColors.push(LightSourceTransfer.push(lightSourceMaterialIndices.push(lightSourceMaterialIndices.push(
```
Looking ahead

- Further reducing MCJIT memory footprint
- Better error handling in out-of-memory scenarios
- LLDB on Windows
- Clang on Windows
- GPU debugging?