Meta-Programming and JIT Compilation

Sean Treichler

Portability vs. Performance

- **Many scientific codes spend ~100% of their cycles in a tiny fraction of the code base**

- **We want these kernels to be as fast as possible, so we:**
  - **Start with an efficient algorithm**
  - **Rely on the compiler's help to optimize the code**
  - **Manually perform compiler-like optimizations (e.g. loop unrolling)**
  - **Take advantage of processor-specific features (e.g. SIMD)**
  - **Add prefetching, block transfers, etc. to improve memory BW**
  - **Inline/fold values that are constant at compile-time**
  - **Optimize for a known memory layout**
  - **Hoist out computations based on run-time parameters that change slowly/not at all**
  - **Tune them based on run-time profiling**
  - **...**

The Problem(s)

- Each additional step generally improves performance, but:
  - Decreases portability
- Can write multiple versions to target different machines, use cases:
  - Increases code devel/debug/maintenance costs
- Optimizations are baked into the checked-in source:
  - Obfuscates intent of code

A Solution: Meta-Programming

- Instead of writing (many variations of) your kernel:
  - Write code that generates the variations programmatically
- Ideally write kernels at a high level, focusing on intent
- Apply target-/use-case-specific optimizations by lowering code through layers of abstraction
  - Code transformed programmatically, at compile time
  - Provides benefits of abstraction, without runtime overhead
  - Transformations themselves are often applicable to many types of kernels

Meta-Programming isn't New

- Meta-Programming exists in many forms today:
  - Offline – app-specific code generators (e.g. Singe, FFTW)
  - Compile-time – e.g. C++ templates
  - Run-time – e.g. Lisp, MetaOCaml
- Legion applications already meta-programming offline, at compile-time
- Would like to meta-program at runtime, in a way that:
  - Generates FAST code
  - Takes advantage of Legion runtime information

Introducing Lua-Terra

- An active research project at Stanford

  http://terralang.org

- Starts with Lua:
  - a very simple dynamic scripting language
  - designed in late ‘90s, fairly “mature” at this point
  - designed to be embeddable just about anywhere

- And then adds Terra:
  - a statically typed, just-in-time (JIT) compiled language
  - designed to interoperate with Lua code
  - also designed to interoperate with existing compiled code

Introduction to Lua-Terra

```
function lua_addone(a)
  return a + 1
end
```

simple Lua function adds to whatever it's given

```
> = lua_addone(10)
11
```

evaluated in Lua's stack-based VM

```
terra terra_addone(a : int) : int
  return a + 1
end
```

Terra looks a lot like Lua code, except with types

```
> = terra_addone(10)
11
```

function is compiled to native code using LLVM, executed directly on host CPU

Capturing JIT-time Constants

```
> X, Y = 10, 100
```

arbitrary Lua variables

```
terra foo(a : int) : int
  for i = 0,X do
    a = a + Y
  end
end
```

captured when Terra function is defined

```
> foo:disas()
...
assembly for function at address 0x22e6070
0x22e6070(+0):     lea    EAX, DWORD PTR [EDI + 1000]
0x22e6077(+7):     ret
```

allowing the compiler to optimize a specialized version

Functions, Types are Lua Objects

```
function axpy(T)
  return terra(alpha : T, X : &T, Y : &T, n : int)
    for i=0,n do
      Y[i] = Y[i] + alpha * X[i]
    end
  end
end

saxpy = axpy(float)
daxpy = axpy(double)

caxpy = axpy(Complex(float))
zaxpy = axpy(Complex(double))
```

Lua function takes a type as a parameter

defines an anonymous Terra function, using the in-scope Lua variable for type

base Terra types are just Lua values, functions returned by "generator" are given useful names

types themselves can be generated by other Lua code

Quotes, Escapes

```
function spmv(A)
  local function body(y, x)
    local assns = {}
    for i = 1,A.rows do
      local sum = `0
      for _,nz in pairs(A[i]) do
        local col, weight = nz[1], nz[2]
        sum = `sum + weight * x[col]
      end
      assns[i] = quote y[i-1] = sum end
    end
    return assns
  end
  return terra(y : &double, x : &double)
    [ body(y, x) ]
  end
end
```

Lua code looks at structure of sparse matrix at invocation

iterates to generate a quoted Terra expression for each row's sum

returns the sequence of quoted Terra statements a list

escape from Terra to Lua to generate list, interpolate statements into Terra function

Portability, Dynamic Tasks

- Terra generates LLVM IR/bitcode – can target:
  - x86 (+SSE, AVX, AVX512, ...)
  - CUDA
  - ARM
  - anything else for which an LLVM backend exists

- Expanding Legion task registration API
  - Tasks can be dynamically registered during execution
  - Take advantage of properties of program input
  - Registration can specify constraints on usage
    - Preserves mapper's ability to make "arbitrary" decisions

On-Demand Variant Generation

- Recall that multiple variants of a task can be registered
  - Runtime will select a variant that is compatible with the processor, instance layouts chosen by the mapper
  - Don't really want to pre-generate all possible variants though...

- Instead register a "variant generator" function
  - Generator function is written in Lua
  - Will be called by the runtime if no suitable variant exists
  - Runtime provides the processor/layout information
  - Generator function returns a new task variant and conditions under which it can be used

Meta-Programming within Legion

- Planning to take advantage of meta-programming within runtime as well
- DMA Subsystem
  - Exponential explosion of memory types, instance layouts
  - Could even specialize for particular index space sparsity
- Avoiding compile-time capacity limits
  - e.g. number of fields per instance
- Dynamic optimization of dependency analysis
  - next step after trace replay

Beyond Lua-Terra

- Modular architecture - Terra is just the trailblazer
- Task registration API supports different “languages”
  - C function pointer
  - Terra expression
  - name of symbol from dynamic shared object
  - LLVM IR
  - ...
- Works for variant generators as well
  - Lua
  - native C/C++?
  - queries to a remote database?