

Terra: A Multi-Stage Language for High-Performance Computing

Presenter: Cunyuan

February 23, 2023

Goals

- Performance matters!

Goals

- Performance matters!
- Low-level languages(e.g. C) are good: we need to make best use of features of the target architecture(e.g. vector instructions).

Goals

- Performance matters!
- Low-level languages(e.g. C) are good: we need to make best use of features of the target architecture(e.g. vector instructions).
- Programming is difficult!

Goals

- Performance matters!
- Low-level languages(e.g. C) are good: we need to make best use of features of the target architecture(e.g. vector instructions).
- Programming is difficult!
- Solution: use high-level languages to generate low-level languages code(e.g. FFTW: OCaml \rightarrow C).

New Problems

- In this case, we get three components.

New Problems

- In this case, we get three components.
- Optimizer: generate plan to guide how to generate code.

New Problems

- In this case, we get three components.
- Optimizer: generate plan to guide how to generate code.
- Compiler: generate target code based on the plan.

New Problems

- In this case, we get three components.
- Optimizer: generate plan to guide how to generate code.
- Compiler: generate target code based on the plan.
- Runtime: support the generated code and provide feedback to the optimizer.

New Problems

- In this case, we get three components.
- Optimizer: generate plan to guide how to generate code.
- Compiler: generate target code based on the plan.
- Runtime: support the generated code and provide feedback to the optimizer.
- Problem1: Separating compiler and optimizer from the runtime makes it difficult to feed runtime statistics back to the compiler to perform problem-specific optimizations.

New Problems

- In this case, we get three components.
- Optimizer: generate plan to guide how to generate code.
- Compiler: generate target code based on the plan.
- Runtime: support the generated code and provide feedback to the optimizer.
- Problem1: Separating compiler and optimizer from the runtime makes it difficult to feed runtime statistics back to the compiler to perform problem-specific optimizations.
- Problem2: How can we re-use legacy libraries?

Two-Language Design

- Lua: high-level, dynamically typed, automatic mm, first class functions.

Two-Language Design

- Lua: high-level, dynamically typed, automatic mm, first class functions.
- Terra(new!): statically typed, manual mm.

Two-Language Design

- Lua: high-level, dynamically typed, automatic mm, first class functions.
- Terra(new!): statically typed, manual mm.
- Use Lua to manipulate Terra code.

Two-Language Design

- Lua: high-level, dynamically typed, automatic mm, first class functions.
- Terra(new!): statically typed, manual mm.
- Use Lua to manipulate Terra code.
- Shared lexical scoping, which is hygienic.

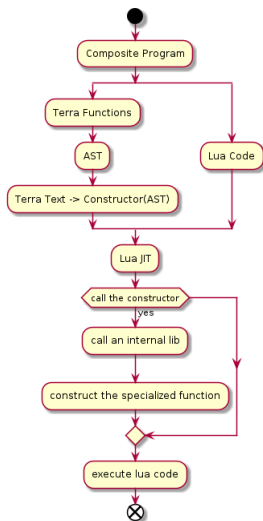
Two-Language Design

- Lua: high-level, dynamically typed, automatic mm, first class functions.
- Terra(new!): statically typed, manual mm.
- Use Lua to manipulate Terra code.
- Shared lexical scoping, which is hygienic.
- Terra code runs independently, to avoid including high-level features.

Two-Language Design

- Lua: high-level, dynamically typed, automatic mm, first class functions.
- Terra(new!): statically typed, manual mm.
- Use Lua to manipulate Terra code.
- Shared lexical scoping, which is hygienic.
- Terra code runs independently, to avoid including high-level features.
- Lua's stack-based C API makes it easy to interface with legacy code.

Two-Language Design



Some Code Examples

```
terra min(a: int, b: int): int
  if a < b then return a
  else return b end
end
struct GreyScaleImage {
  data: &float
  N: int
}
```

Features

- Terra entities are all first-class Lua values.

Features

- Terra entities are all first-class Lua values.
- Terra functions will be executed in LLVM JIT.

Features

- Terra entities are all first-class Lua values.
- Terra functions will be executed in LLVM JIT.
- You can dump Terra functions to an object file(i.e. something.o in Linux) if you like.

Features

- Terra entities are all first-class Lua values.
- Terra functions will be executed in LLVM JIT.
- You can dump Terra functions to an object file(i.e. something.o in Linux) if you like.
- Quotation: using brackets([]) for escaping and backtick(expressions)/quote keyword(statements) for creating quotation.

Quotation Example

```
local a = 5
terra sin5()
  return [ math.sin(a) ]
end
function addtwo(a,b)
  return 'a + b'
end
local printtwice = quote
  C.printf("hello\n")
  C.printf("hello\n")
end
```


It Just Works!

```
-- C++                                -- Lua/Terra
int add(int a, int b) {               | terra add(a : int, b : int) : int
    return a + b;                     |     return a + b
}                                     |     end

                                     |
                                     | -- Conditional compilation is done
                                     | -- with control-flow that
                                     | -- determines what code is defined
#ifdef _WIN32                         | if iswindows() then
void waitatend() {                   |     terra waitatend()
    getchar();                       |     C.getchar()
}                                     |     end
#else                                | else
void waitatend() {}                 |     terra waitatend() end
#endif                               | end

                                     |
                                     | -- Templates become Lua functions
                                     | -- that take a terra type T and
                                     | -- use it to generate new types
                                     | -- and code
template<class T>                   | function Array(T)
struct Array {                       |     struct Array {
    int N;                           |         N : int
    T* data;                         |         data : &T
    }
    T get(int i) {                  |     terra Array:get(i : int)
        return data[i];             |         return self.data[i]
    }                               |     end
};                                   |     return Array
typedef                             | end
Array<float> FloatArray;             | FloatArray = Array(float)
```

It Just Works!

- Now we can generate code dynamically.

It Just Works!

- Now we can generate code dynamically.
- e.g. block the loop nests to make the memory access more friendly to the cache.

A Simple Filter Example

```
GreyscaleImage = Image(float)
terra laplace(img: &GreyscaleImage,
out: &GreyscaleImage) : {}
  —shrink result, do not calculate boundaries
  var newN = img.N - 2
  out: init(newN)
  for i = 0, newN do
    for j = 0, newN do
      var v = img: get(i+0, j+1) + img: get(i+2, j+1)
        10 + img: get(i+1, j+2) + img: get(i+1, j+0)
        - 4 * img: get(i+1, j+1)
      out: set(i, j, v)
    end
  end
end
```

A Simple Filter Example

```
terra runlaplace(input: rawstring ,  
output: rawstring) : {}  
  var i = GreyscaleImage {}  
  var o = GreyscaleImage {}  
  i:load(input)  
  laplace(&i,&o)  
  o:save(output)  
  i:free(); o:free()  
end
```

Optimize The Filter

```
function blockedloop(N,blocksizes,bodyfn)
  local function generatelevel(n,ii,jj,bb)
    if n > #blocksizes then
      return bodyfn(ii,jj)
    end
    local blocksize = blocksizes[n]
    return quote
      for i = ii,min(ii+bb,N),blocksize do
        for j = jj,min(jj+bb,N),blocksize do
          [ generatelevel(n+1,i,j,blocksize) ]
        end
      end
    end
  end
  return generatelevel(1,0,0,N)
end
```

Optimize The Filter

```
GreyscaleImage = Image(float)
terra laplace(img: &GreyscaleImage,
out: &GreyscaleImage) : {}
  —shrink result, do not calculate boundaries
  var newN = img.N - 2
  out: init(newN)
  [blockedloop(newN, {128, 64, 1}, function(i, j)
    return quote
      var v = img: get(i+0, j+1) + img: get(i+2, j+1)
      + img: get(i+1, j+2) + img: get(i+1, j+0)
      - 4 * img: get(i+1, j+1)
      out: set(i, j, v)
    end
  end)]
end
```

The Formal Calculus: Terra Core

- For simplicity, Lua := imperative language + first-class functions,
Terra := purely functional language

The Formal Calculus: Terra Core

- For simplicity, Lua := imperative language + first-class functions, Terra := purely functional language
- Lua expression: e , evaluation of Lua: \xrightarrow{L}

The Formal Calculus: Terra Core

- For simplicity, Lua := imperative language + first-class functions, Terra := purely functional language
- Lua expression: e , evaluation of Lua: \xrightarrow{L}
- Terra expression: \hat{e} , specialization of Terra: \xrightarrow{S}

The Formal Calculus: Terra Core

- For simplicity, Lua := imperative language + first-class functions, Terra := purely functional language
- Lua expression: e , evaluation of Lua: \xrightarrow{L}
- Terra expression: \dot{e} , specialization of Terra: \xrightarrow{S}
- Specialized Terra expression: $\underline{\dot{e}}$, execution of specialized Terra expression: \xrightarrow{T}

Lua Syntax:

$$\begin{aligned} e &::= b \mid \dot{T} \mid x \mid \text{let } x = e \text{ in } e \mid x := e \mid e(e) \mid \text{fun}(x)\{e\} \mid tdecl \mid \\ &\quad \text{ter } e(x : e) : e\{\dot{e}\} \mid \backslash \dot{e} \\ v &::= b \mid l \mid \dot{T} \mid \langle \Gamma, x, e \rangle \mid \underline{\dot{e}} \\ \dot{T} &::= \dot{B} \mid \dot{T} \rightarrow \dot{T} \end{aligned}$$

Terra Syntax:

$$\dot{e} ::= b \mid x \mid \dot{e}(\dot{e}) \mid \text{tlet } x: e = \dot{e} \text{ in } \dot{e} \mid [e]$$
$$\underline{\dot{e}} ::= b \mid \underline{\dot{x}} \mid \underline{\dot{e}}(\underline{\dot{e}}) \mid \text{tlet } \underline{\dot{x}}: \dot{T} = \underline{\dot{e}} \text{ in } \underline{\dot{e}} \mid I$$

$$v \Sigma \xrightarrow{L} v \Sigma \quad (\text{LVAL})$$

$$\frac{\Sigma = \Gamma, S, F}{x \Sigma \xrightarrow{L} S(\Gamma(x)) \Sigma} \quad (\text{LVAR})$$

$$\frac{e_1 \Sigma_1 \xrightarrow{L} v_1 \Sigma_2 \quad \Sigma_2 = \Gamma, S, F \quad e_2 \Sigma_2[x \leftarrow v_1] \xrightarrow{L} v_2 \Sigma_3}{\text{let } x = e_1 \text{ in } e_2 \Sigma \xrightarrow{L} v_2(\Sigma_3 \leftarrow \Gamma)} \quad (\text{LLET})$$

$$\frac{e \Sigma \xrightarrow{L} v \Gamma, S, F \quad \Gamma(x) = a}{x := e \Sigma \xrightarrow{L} v \Gamma, S[a \leftarrow v], F} \quad (\text{LASN})$$

$$\frac{\Sigma = \Gamma, S, F}{\text{fun}(x)\{e\} \Sigma \xrightarrow{L} \langle \Gamma, x, e \rangle \Sigma} \quad (\text{LFUN})$$

$$\frac{e_1 \Sigma_1 \xrightarrow{L} \langle \Gamma_1, x, e_3 \rangle \quad e_2 \Sigma_2 \xrightarrow{L} v_1 \Gamma_2, S, F \quad a \text{ fresh} \quad e_3 \Gamma_1[x \leftarrow a], S[a \leftarrow v_1], F \xrightarrow{L} v_2 \Sigma_3}{e_1(e_2) \Sigma_1 \xrightarrow{L} v_2 (\Sigma_3 \leftarrow \Gamma_2)} \quad (\text{LAPP})$$

$$\frac{l \text{ fresh} \quad \Sigma = \Gamma, S, F}{\text{tdec} \Sigma \xrightarrow{L} l \Gamma, S, F[l \leftarrow \bullet]} \quad (\text{LTDECL})$$

$$\begin{array}{c}
 e_1 \Sigma_1 \xrightarrow{L} I \Sigma_2 \quad e_2 \Sigma_2 \xrightarrow{L} \dot{T}_1 \Sigma_3 \quad e_3 \Sigma_3 \xrightarrow{L} \dot{T}_2 \Sigma_4 \\
 \Sigma_4 = \Gamma_1, S_1, F_1 \quad \dot{x} \text{ fresh} \\
 \frac{\dot{e} \Sigma_4[x \leftarrow \dot{x}] \xrightarrow{S} \dot{x} \Gamma_2, S_2, F_2 \quad F_2(I) = \bullet}{\text{ter } e_1(x : e_2) : e_3\{\dot{e}\} \Sigma_1 \xrightarrow{L} I \Gamma_1, S_2, F_2[I \leftarrow \langle \dot{x}, \dot{T}_1, \dot{T}_2, \dot{e} \rangle]} \quad (\text{LTDEFN})
 \end{array}$$

$$\frac{\dot{e} \Sigma_1 \xrightarrow{S} \dot{e} \Sigma_2}{\backslash \dot{e} \Sigma_1 \xrightarrow{L} \dot{e} \Sigma_2} \quad (\text{LTQUOTE})$$

$$\begin{array}{c}
 e_1 \Sigma_1 \xrightarrow{L} l \Sigma_2 \quad e_2 \Sigma_2 \xrightarrow{L} b_1 \Sigma_3 \\
 \Sigma_3 = \Gamma, S, F \quad F(l) = \langle \dot{x}, \dot{T}_1, \dot{T}_2, \dot{e} \rangle \quad b_1 \in \dot{T}_1 \\
 \frac{[\dot{x} : \dot{T}_1], [l : \dot{T}_1 \rightarrow \dot{T}_2], F_2 \vdash \dot{e} : \dot{T}_2 \quad \dot{e}[\dot{x} \leftarrow b], F \xrightarrow{T} b_2}{e_1(e_2) \Sigma_1 \xrightarrow{L} b_2 \Sigma_3} \quad (\text{LTAPP})
 \end{array}$$

$$b \Sigma \xrightarrow{S} b \Sigma \quad (\text{SBAS})$$

$$\frac{\dot{e}_1 \Sigma_1 \xrightarrow{S} \underline{\dot{e}_1} \Sigma_2 \quad \dot{e}_2 \Sigma_2 \xrightarrow{S} \underline{\dot{e}_2} \Sigma_3}{\dot{e}_1(\dot{e}_2) \Sigma_1 \xrightarrow{S} \underline{\dot{e}_1}(\underline{\dot{e}_2}) \Sigma_3} \quad (\text{SAPP})$$

$$\frac{e \Sigma_1 \xrightarrow{L} \dot{T} \Sigma_2 \quad \dot{e}_1 \Sigma_2 \xrightarrow{S} \underline{\dot{e}_1} \Sigma_3 \quad \underline{\dot{x}} \text{ fresh} \quad \Sigma_3 = \Gamma, S, F \quad \dot{e}_2 \Sigma_3[x \leftarrow \underline{\dot{e}_2}] \xrightarrow{S} \underline{\dot{e}_2} \Sigma_4}{tlet \ x : e = \dot{e}_1 \text{ in } \dot{e}_2 \Sigma_1 \xrightarrow{S} tlet \ \underline{\dot{x}} : \dot{T} = \underline{\dot{e}_1} \text{ in } \underline{\dot{e}_2} (\Sigma_4 \leftarrow \Gamma)} \quad (\text{SLET})$$

$$\frac{e \Sigma_1 \xrightarrow{L} \underline{e} \Sigma_2}{[e] \Sigma_1 \xrightarrow{S} \underline{e} \Sigma_2} \quad (\text{SESC})$$

$$\frac{[x] \Sigma_1 \xrightarrow{S} \underline{e} \Sigma_2}{x \Sigma_1 \xrightarrow{S} \underline{e} \Sigma_2} \quad (\text{SVAR})$$

$$b \dot{\Gamma}, F \xrightarrow{T} b \quad (\text{TBAS})$$

$$l \dot{\Gamma}, F \xrightarrow{T} l \quad (\text{TFUN})$$

$$\dot{x} \dot{\Gamma}, F \xrightarrow{T} \dot{\Gamma}(\dot{x}) \quad (\text{TVAR})$$

$$\frac{\underline{e_1} \dot{\Gamma}, F \xrightarrow{T} v_1 \quad \underline{e_2} \dot{\Gamma}[\dot{x} \leftarrow v_1], F \xrightarrow{T} v_2}{tlet \dot{x} : \dot{T} = \underline{e_1} \text{ in } \underline{e_2} \dot{\Gamma}, F \xrightarrow{T} v_2} \quad (\text{TLET})$$

$$\frac{\underline{e_1} \dot{\Gamma}, F \xrightarrow{T} l \quad \underline{e_2} \dot{\Gamma}, F \xrightarrow{T} v_1 \quad F(l) = \langle \dot{x}, \dot{T}_1, \dot{T}_2, \underline{e_3} \rangle \quad \underline{e_3} \dot{\Gamma}[\dot{x} \leftarrow v_1], F \xrightarrow{T} v_2}{\underline{e_1}(\underline{e_2}) \dot{\Gamma}, F \xrightarrow{T} v_2} \quad (\text{TAPP})$$

- The typing rules are very simple. Skip.

- The typing rules are very simple. Skip.
- Let's see the proof.

- The typing rules are very simple. Skip.
- Let's see the proof.
- No proof! :)

Some Important Designs

- Use shared lexical environments to reduce the need for escape expressions.

Some Important Designs

- Use shared lexical environments to reduce the need for escape expressions.
- Perform specialization eagerly.

Some Important Designs

- Use shared lexical environments to reduce the need for escape expressions.
- Perform specialization eagerly.
- Perform typechecking and linking lazily.

Some Important Designs

- Use shared lexical environments to reduce the need for escape expressions.
- Perform specialization eagerly.
- Perform typechecking and linking lazily.
- Type Reflection API.

Why Specialize Eagerly?

```
let  $x_1 = 0$   
let  $y = \text{ter } tdecl(x_2 : \text{int}) : \text{int } \{x_1\}$  in  
 $x_1 := 1; y(0)$ 
```

Why Specialize Eagerly?

```
let  $x_1 = 0$   
let  $y = \text{ter } tdecl(x_2 : \text{int}) : \text{int } \{x_1\}$  in  
 $x_1 := 1; y(0)$ 
```

- $y(0) = 0$ if we specialize eagerly.

Why Specialize Eagerly?

```
let  $x_1 = 0$   
let  $y = \text{ter tdecl}(x_2 : \text{int}) : \text{int} \{x_1\}$  in  
 $x_1 := 1; y(0)$ 
```

- $y(0) = 0$ if we specialize eagerly.
- If not, the Terra runtime must depends on the Lua runtime to get correct value of x_1 .

Why Specialize Eagerly?

```
let  $x_1 = 0$   
let  $y = \text{ter tdecl}(x_2 : \text{int}) : \text{int} \{x_1\}$  in  
 $x_1 := 1; y(0)$ 
```

- $y(0) = 0$ if we specialize eagerly.
- If not, the Terra runtime must depend on the Lua runtime to get correct value of x_1 .
- Or we need to re-compile y when x_1 changes.

Why Specialize Eagerly?

```
let  $x_1 = 0$   
let  $y = \text{ter } tdecl(x_2 : \text{int}) : \text{int } \{x_1\} \text{ in}$   
 $x_1 := 1; y(0)$ 
```

- $y(0) = 0$ if we specialize eagerly.
- If not, the Terra runtime must depend on the Lua runtime to get correct value of x_1 .
- Or we need to re-compile y when x_1 changes.
- Requires declaration before using a symbol, which makes recursive function impossible.

Why Specialize Eagerly?

```
let  $x_1 = 0$   
let  $y = \text{ter } tdecl(x_2 : \text{int}) : \text{int } \{x_1\} \text{ in}$   
 $x_1 := 1; y(0)$ 
```

- $y(0) = 0$ if we specialize eagerly.
- If not, the Terra runtime must depend on the Lua runtime to get correct value of x_1 .
- Or we need to re-compile y when x_1 changes.
- Requires declaration before using a symbol, which makes recursive function impossible.
- Separate the declaration and definition.

Why Typecheck Lazily?

- Possible if we use type annotations.

Why Typecheck Lazily?

- Possible if we use type annotations.
- Since a function may have no definition, this may not help.

Why Typecheck Lazily?

- Possible if we use type annotations.
- Since a function may have no definition, this may not help.
- So no need for type annotations.

Why Typecheck Lazily?

- Possible if we use type annotations.
- Since a function may have no definition, this may not help.
- So no need for type annotations.
- Easier to override the default behavior of a type.

Type Reflection

- Terra types are first-class in Lua.

Type Reflection

- Terra types are first-class in Lua.
- Provide methods in Lua(e.g. `t:ispointer` or `t:isstruct`).

Type Reflection

- Terra types are first-class in Lua.
- Provide methods in Lua(e.g. `t:ispointer` or `t:isstruct`).
- Use entries table to describe structures' in-memory layout.

```
struct Complex {}  
Complex.entries:insert { field = "real",  
    type = float }  
Complex.entries:insert { field = "imag",  
    type = float }
```


Type Reflection

- Terra types are first-class in Lua.
- Provide methods in Lua(e.g. `t:ispointer` or `t:isstruct`).
- Use entries table to describe structures' in-memory layout.

```
struct Complex {  
  Complex.entries:insert { field = "real",  
    type = float }  
  Complex.entries:insert { field = "imag",  
    type = float }
```

- Also the metamethods table: override certain compile-time behaviors(e.g. implicit conversion).

Evaluation

- Similar performance to ATLAS.

Evaluation

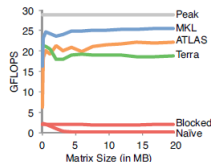
- Similar performance to ATLAS.
- ATLAS: a high-performance scientific computing library written in C and assembly.

Evaluation

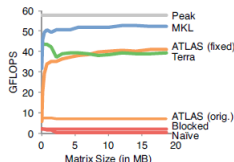
- Similar performance to ATLAS.
- ATLAS: a high-performance scientific computing library written in C and assembly.
- Shorter code, easier to read/write/maintain.

Evaluation

- Similar performance to ATLAS.
- ATLAS: a high-performance scientific computing library written in C and assembly.
- Shorter code, easier to read/write/maintain.



(a) DGEMM Performance



(b) SGEMM Performance

Summary

- Two-Languages design: Lua + Terra.

Summary

- Two-Languages design: Lua + Terra.
- Shared lexical scoping.

Summary

- Two-Languages design: Lua + Terra.
- Shared lexical scoping.
- Seperate Evaluationn: Lua(LuaJIT), Terra(LLVM JIT).

Summary

- Two-Languages design: Lua + Terra.
- Shared lexical scoping.
- Seperate Evaluationn: Lua(LuaJIT), Terra(LLVM JIT).
- Type Reflection.

Advantages

- No need to write C, but the performance is still good.

Advantages

- No need to write C, but the performance is still good.
- Generating code dynamically allows us to use runtime information from LuaJIT.

Advantages

- No need to write C, but the performance is still good.
- Generating code dynamically allows us to use runtime information from LuaJIT.
- Easy to re-use C/C++ libraries in Lua.

Advantages

- No need to write C, but the performance is still good.
- Generating code dynamically allows us to use runtime information from LuaJIT.
- Easy to re-use C/C++ libraries in Lua.

Shortages

- Lua is not statically typed.

Reference

- Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. 2013. Terra: A multi-stage language for highperformance computing. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13). ACM, New York, 105–116.
DOI:<https://doi.org/10.1145/2491956.2462166>
- Terra: A low-level counterpart to Lua. <https://terralang.org/>