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

Zachary DeVito[1] James Hegarty[1] Alex Aiken[1] Pat Hanrahan[1] Jan Vitek[2]

[1]Stanford University [2]Purdue University

February 16, 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 taget 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 taget 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 taget code based on the plan.
- Runtime: support the generated code and provide feedback to the optimizer.
- Problem1: How can we get the runtime statistics in the compiler and generate high-performance code dynamically?

New Problems

- In this case, we get three components.
- Optimizer: generate plan to guide how to generate code.
- Compiler: generate taget code based on the plan.
- Runtime: support the generated code and provide feedback to the optimizer.
- Problem1: How can we get the runtime statistics in the compiler and generate high-performance code dynamically?
- 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, manumal mm.

Two-Language Design

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

Two-Language Design

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

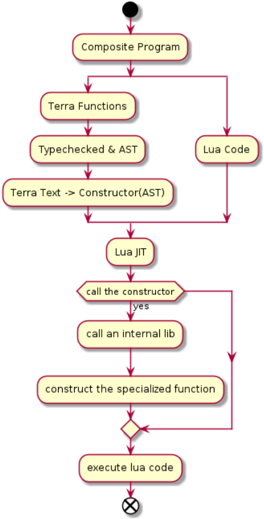
Two-Language Design

- Lua: high-level, dynamically typed, automatic mm, first class functions.
- Terra(new!): statically typed, manumal 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, manumal 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
};                            |  end
typedef                       |
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.

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: $\dot{e}$, 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 specailized Terra expression: $\xrightarrow{T}$

Terra Core

Lua Syntax:

$$
\begin{aligned}
e &::= b \mid \dot{T} \mid x \mid let\, x = e\, in\, e \mid x := e \mid e(e) \mid fun(x)\{e\} \mid tdecl \mid \\
&\quad tere(x : e) : e\{\dot{e}\} \mid \text{`}\dot{e} \\
v &::= b \mid l \mid \dot{T} \mid <\Gamma, x, e> \mid \underline{\dot{e}} \\
\dot{T} &::= \dot{B} \mid \dot{T} \rightarrow \dot{T}
\end{aligned}
$$

Terra Core

Terra Syntax:

$$\dot{e} ::= b \mid x \mid \dot{e}(\dot{e}) \mid \mathit{tlet}\, x :\ e = \dot{e}\ \mathit{in}\ \dot{e} \mid [e]$$
$$\underline{\dot{e}} ::= b \mid \underline{\dot{x}} \mid \underline{\dot{e}}(\underline{\dot{e}}) \mid \mathit{tlet}\, \underline{\dot{x}} :\ \dot{T} = \underline{\dot{e}}\ \mathit{in}\ \underline{\dot{e}} \mid l$$

Terra Core

$$v\,\Sigma \xrightarrow{L} v\,\Sigma \tag{LVAL}$$

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

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

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

Terra Core

$$\frac{\Sigma = \Gamma, S, F}{fun(x)\{e\}\ \Sigma \xrightarrow{L} < \Gamma, x, e > \ \Sigma} \tag{LFUN}$$

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

$$\frac{l\ fresh\ \ \Sigma = \Gamma, S, F}{tdecl \Sigma \xrightarrow{L} l\, \Gamma, S, F[l \leftarrow \bullet]} \tag{LTDECL}$$

Terra Core

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

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

Terra Core

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

Terra Core

$$b\,\Sigma \xrightarrow{S} b\,\Sigma \tag{SBAS}$$

$$\frac{\dot{e_1}\,\Sigma_1 \xrightarrow{S} \underline{\dot{e_1}}\,\Sigma_2 \quad \dot{e_2}\Sigma_2 \xrightarrow{S} \underline{\dot{e_2}}\Sigma_3}{\dot{e_1}(\dot{e_2})\,\Sigma_1 \xrightarrow{S} \underline{\dot{e_1}}(\underline{\dot{e_2}})\,\Sigma_3} \tag{SAPP}$$

$$\frac{\begin{gathered} e\,\Sigma_1 \xrightarrow{L} \dot{T}\,\Sigma_2 \quad \dot{e_1}\,\Sigma_2 \xrightarrow{S} \underline{\dot{e_1}}\,\Sigma_3 \quad \underline{\dot{x}}\,\mathit{fresh} \\ \Sigma_3 = \Gamma, S, F \quad \dot{e_2}\,\Sigma_3[x \leftarrow \underline{\dot{e_2}}] \xrightarrow{S} \underline{\dot{e_2}}\,\Sigma_4 \end{gathered}}{\mathit{tlet}\,x : e = \dot{e_1}\ \mathit{in}\,\dot{e_2}\,\Sigma_1 \xrightarrow{S} \mathit{tlet}\,\underline{\dot{x}} : \dot{T} = \underline{\dot{e_1}}\ \mathit{in}\,\underline{\dot{e_2}}\,(\Sigma_4 \leftarrow \Gamma)} \tag{SLET}$$

Terra Core

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

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

Terra Core

$$\underline{b}\,\dot{\Gamma}, F \xrightarrow{T} b \tag{TBAS}$$

$$\underline{l}\,\dot{\Gamma}, F \xrightarrow{T} l \tag{TFUN}$$

$$\underline{\dot{x}}\,\dot{\Gamma}, F \xrightarrow{T} \dot{\Gamma}(\underline{\dot{x}}) \tag{TVAR}$$

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

$$\frac{\begin{array}{l}\underline{\dot{e_1}}\,\dot{\Gamma}, F \xrightarrow{T} l \quad \underline{\dot{e_2}}\,\dot{\Gamma}, F \xrightarrow{T} v_1 \\ F(l) = < \underline{\dot{x}}, \dot{T_1}, \dot{T_2}, \underline{\dot{e_3}} > \quad \underline{\dot{e_3}}\,\dot{\Gamma}[\underline{\dot{x}} \leftarrow v_1], F \xrightarrow{T} v_2\end{array}}{\underline{\dot{e_1}}(\underline{\dot{e_2}})\,\dot{\Gamma}, F \xrightarrow{T} v_2} \tag{TAPP}$$

Terra Core

- The typing rules are very simple. Skip.

Terra Core

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

Terra Core

- 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?

$$
\begin{aligned}
&let\, x_1 \;=\; 0\\
&let\, y \;=\; ter\, tdecl(x_2 : \; int) : \; int\, \{x_1\}\; in\\
&x_1 \;:=\; 1;\; y(0)
\end{aligned}
$$

Why Specialize Eagerly?

$$\begin{aligned} &let\, x_1 \;=\; 0 \\ &let\, y \;=\; ter\, tdecl(x_2 : int) : int\, \{x_1\}\; in \\ &x_1 \;:=\; 1;\; y(0) \end{aligned}$$

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

Why Specialize Eagerly?

$$\begin{aligned} &let\, x_1 \;=\; 0 \\ &let\, y \;=\; ter\, tdecl(x_2 : int) : int\, \{x_1\}\, in \\ &x_1 \;:=\; 1;\, y(0) \end{aligned}$$

- $y(0) = 0$ 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?

$$
\begin{aligned}
let\, x_1 &= 0 \\
let\, y &= ter\, tdecl(x_2 : int) : int\, \{x_1\}\, in \\
x_1 &:= 1;\ y(0)
\end{aligned}
$$

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

Why Specialize Eagerly?

$$
\begin{aligned}
&let\, x_1 \;=\; 0 \\
&let\, y \;=\; ter\, tdecl(x_2 : \; int) : \; int\, \{x_1\}\; in \\
&x_1 \;:=\; 1;\; y(0)
\end{aligned}
$$

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

Why Specialize Eagerly?

$$
\begin{aligned}
&let\, x_1 \;=\; 0 \\
&let\, y \;=\; ter\, tdecl(x_2 :\; int) :\; int\, \{x_1\}\; in \\
&x_1 \;:=\; 1;\; y(0)
\end{aligned}
$$

- $y(0) = 0$ if we specialize eagerly.
- If not, the Terra runtime must depends on the Lua runtime to get correct value of x_1.
- Or we need to re-compile y when x_1 changes.
- Requires declaration before using a symbol, which makes recusive 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 bevavior 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, which is written in C and assembly.

Evaluation

- Similar performance to ATLAS, which is written in C and assembly.
- Shorter code, easier to read/write/maintain.

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/