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.

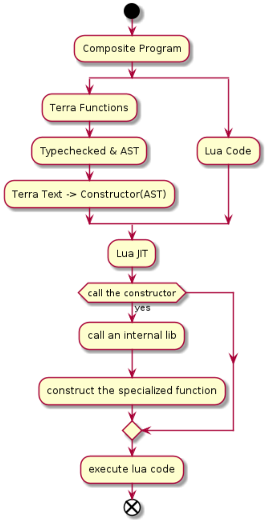
Two-Language Design

- 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.

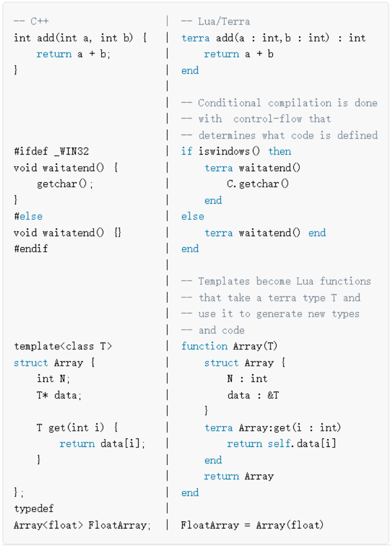
Features

- 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:

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

Terra Core

Terra Syntax:

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

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

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

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

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

Terra Core

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

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

Terra Core

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

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.