The Design of Terra
Harnessing the best features of high-level and low-level languages to build DSLs

Zach DeVito      Pat Hanrahan
Stanford University
In some domains, we don’t want high-performance, we want **peak** performance:

**Rendering** [Foley et al. ’11]

**Physical Simulation** [Bernstein et al. ’15]

**Image Processing** [Hegarty et al. ’14]

**Optimization** [Lin et al. ’13, Nießner et al. ’14]
Even idiomatic C++ code is inefficient for these domains:

```cpp
void box_filter_3x3(const Image &in, Image &blury) {
    Image blurx(in.width(), in.height()); // allocate blurx array

    for (int y = 0; y < in.height(); y++)
        for (int x = 0; x < in.width(); x++)
            blurx(x,y) = (in(x-1,y) + in(x,y) + in(x+1,y))/3;

    for (int y = 0; y < in.height(); y++)
        for (int x = 0; x < in.width(); x++)
            blury(x,y) = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1))/3;
}
```

[example courtesy of Jonathan Ragan-Kelley]
The optimized implementation is 11x faster

```c
void box_filter_3x3(const Image &in, Image &blury) {
    __m128i one_third = _mm_set1_epi16(21846);
    #pragma omp parallel for
    for (int yTile = 0; yTile < in.height(); yTile += 32) {
        __m128i a, b, c, sum, avg;
        __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurxPtr = blurx;
            for (int y = -1; y < 32+1; y++) {
                const uint16_t *inPtr = &(in[yTile+y][xTile]);
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_loadu_si128((__m128i *)(inPtr-1));
                    b = _mm_loadu_si128((__m128i *)(inPtr+1));
                    c = _mm_load_si128((__m128i *)(inPtr));
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(blurxPtr++, avg);
                }
                inPtr += 8;
            }
            blurxPtr = blurx;
        }
    }
}```
The optimized implementation is 11x faster

```c
void box_filter_3x3(const Image &in, Image &blury) {
    __m128i one_third = _mm_set1_epi16(21846);
    #pragma omp parallel for
    for (int yTile = 0; yTile < in.height(); yTile += 32) {
        __m128i a, b, c, sum, avg;
        __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurxPtr = blurx;
            for (int y = -1; y < 32+1; y++) {
                const uint16_t *inPtr = &(in[yTile+y][xTile]);
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_loadu_si128((__m128i*)(inPtr-1));
                    b = _mm_loadu_si128((__m128i*)(inPtr+1));
                    c = _mm_load_si128((__m128i*)(inPtr));
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(blurxPtr++, avg);
                    inPtr += 8;
                }
                blurxPtr = blurx;
            }
            __m128i *outPtr = (__m128i *)(blury[yTile+y][xTile]);
            for (int x = 0; x < 256; x += 8) {
                a = _mm_load_si128(blurxPtr+(2*256)/8);
                b = _mm_load_si128(blurxPtr+256/8);
                c = _mm_load_si128(blurxPtr++);
                sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                avg = _mm_mulhi_epi16(sum, one_third);
                _mm_store_si128(outPtr++, avg);
            }
        }
    }
}
```

Parallelism
threaded w/ openMP
SIMD vectors
The optimized implementation is 11x faster

```c
void box_filter_3x3(const Image &in, Image &blury) {
    __m128i one_third = _mm_set1_epi16(21846);
    #pragma omp parallel for
    for (int yTile = 0; yTile < in.height(); yTile += 32) {
        __m128i a, b, c, sum, avg;
        __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurxPtr = blurx;
            for (int y = -1; y < 32+1; y++) {
                const uint16_t *inPtr = &(in[yTile+y][xTile]);
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_loadu_si128((__m128i*)(inPtr-1));
                    b = _mm_loadu_si128((__m128i*)(inPtr+1));
                    c = _mm_load_si128((__m128i*)(inPtr));
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(blurxPtr++, avg);
                    inPtr += 8;
                }
                blurxPtr = blurx;
            }
            __m128i *outPtr = (__m128i *)&(blury[yTile+y][xTile]);
            for (int x = 0; x < 256; x += 8) {
                a = _mm_load_si128(blurxPtr+(2*256)/8);
                b = _mm_load_si128(blurxPtr+256/8);
                c = _mm_load_si128(blurxPtr++);
                sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                avg = _mm_mulhi_epi16(sum, one_third);
                _mm_store_si128(outPtr++, avg);
            }
        }
    }
}
```

Parallelism threaded w/ openMP
SIMD vectors

Locality
tiled, intermediate stored in cache
The optimized implementation is 11x faster

```c
void box_filter_3x3(const Image &in, Image &blury) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < in.height(); yTile += 32) {
    __m128i a, b, c, sum, avg;
    __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurxPtr = blurx;
      for (int y = -1; y < 32+1; y++) {
        const uint16_t *inPtr = &(in[yTile+y][xTile]);
        for (int x = 0; x < 256; x += 8) {
          a = _mm_loadu_si128((__m128i*)inPtr-1);
          b = _mm_loadu_si128((__m128i*)inPtr+1);
          c = _mm_load_si128((__m128i*)inPtr);
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(blurxPtr++, avg);
          inPtr += 8;
        }
        blurxPtr = blurx;
      }
      __m128i *outPtr = ((__m128i *)(&blury[yTile][xTile]));
      for (int x = 0; x < 256; x += 8) {
        a = _mm_load_si128(blurxPtr+(2*256)/8);
        b = _mm_load_si128(blurxPtr+256/8);
        c = _mm_load_si128(blurxPtr++);
        sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
        avg = _mm_mulhi_epi16(sum, one_third);
        _mm_store_si128(outPtr++, avg);
      }
    }
}
```

Hard to write. Specialized to one machine.
Specify the operation in a high-level language, then transform it into code in a low-level language.
We already use DSLs to do this transformation

SQL (in Java)
String query = "SELECT * FROM people WHERE age > 18";
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(query);

Regular Expressions (in Python)
>>> p = re.compile(\d+)  
>>> p.findall('12 drummers drumming, 11 pipers piping, 10 lords a-leaping')  
['12', '11', '10']

OpenGL/DirectX (in C/C++)
for(int i = 0; i < 100; i++) {
    glBegin(GL_TRIANGLES);
    glVertex2f(0.0f, 0.0f);
    glVertex2f(1.0f, 0.0f);
    glVertex2f(1.0f, 0.0f);
    glEnd();
    glTranslate2f(1.0f, 0.0f);  
}
Using compilers like LLVM to dynamically generate code tedious and verbose

```c
float solve(float a, float b, float c) {
    return (-b + sqrt(b*b - 4*a*c)) / (2 * a);
}
```
Using compilers like LLVM to dynamically generate code tedious and verbose

```c
float solve(float a, float b, float c) {
    return (-b + sqrt(b*b - 4*a*c)) / (2 * a);
}
```

```c
Value* float_mul = B.CreateFMul(float_b, float_b);
Value* float_mul1 = B.CreateFMul(float_a, const_float_3);
Value* float_mul2 = B.CreateFMul(float_mul1, float_c);
Value* float_sub3 = B.CreateFSub(float_mul, float_mul2);
Value* float_call = B.CreateCall(func_sqrtf, float_sub3);
Value* float_add = B.CreateFSub(float_call, float_b)
Value* float_div = B.CreateFMul(float_add, const_float_4);
Value* float_mul4 = B.CreateFMul(float_div, float_a);
B.CreateReturn(float_mul4);
```
Using compilers like LLVM to dynamically generate code tedious and verbose.

```c++
float solve(float a, float b, float c) {
  return (-b + sqrt(b*b - 4*a*c)) / (2 * a);
}
```

```c++
std::vector<Type*> SolveTy_args;
SolveTy_args.push_back(Type::getFloatTy(C));
SolveTy_args.push_back(Type::getFloatTy(C));
SolveTy_args.push_back(Type::getFloatTy(C));
FunctionType* SolveTy = FunctionType::get(Type::getFloatTy(C), SolveTy_args);

std::vector<Type*> SqrtTy_args;
SqrtTy_args.push_back(Type::getFloatTy(C));
FunctionType* SqrtTy = FunctionType::get(Type::getFloatTy(C), SqrtTy_args);

PointerType* PtrSqrtTy = PointerType::get(SqrtTy, 0);

Function* func_solve = Function::Create(SolveTy, GlobalValue::ExternalLinkage, "solve", M);
Function* func_sqrtf = Function::Create(SqrtTy, GlobalValue::ExternalLinkage, "sqrtf", M);

ConstantFP* const_float_3 = ConstantFP::get(C, 4.f);
ConstantFP* const_float_4 = ConstantFP::get(C, 5.f);

Function::arg_iterator args = func_solve->arg_begin();
Value* float_a = args++;
Value* float_b = args++;
Value* float_c = args++;

BasicBlock* label_entry = BasicBlock::Create(C, "entry", func_solve, 0);
IRBuilder<> * B(label_entry);
Value* float_mul = B.CreateFMul(float_b, float_b);
Value* float_mul1 = B.CreateFMul(float_a, const_float_3);
Value* float_mul2 = B.CreateFMul(float_mul1, float_c);
Value* float_sub3 = B.CreateFSub(float_mul, float_mul2);
Value* float_call = B.CreateCall(func_sqrtf, float_sub3);
Value* float_add = B.CreateFSub(float_call, float_b);
Value* float_div = B.CreateFMul(float_add, const_float_4);
Value* float_mul4 = B.CreateFMul(float_div, float_a);
B.CreateReturn(float_mul4);
```
Our approach: A Two-language design
Designing a Two-language System
Combining High- and Low-level Languages

**Web Server Development**
Database Language (C/C++), ORM layer, Business Logic (Ruby)

**Scientific Computing**
MATLAB, C++/FORTRAN

**Game Programming**
Shading Language (OpenGL), Scripting Language (Lua), Engine Language (C++)
Integrating existing languages is problematic
Integrating existing languages is problematic

Each language makes tradeoffs, adding extra complexity
Integrating existing languages is problematic.

Each language makes tradeoffs, adding extra complexity.
Integrating existing languages is problematic.

Each language makes tradeoffs, adding extra complexity.
Integrating existing languages is problematic. Each language makes tradeoffs, adding extra complexity.
We should design with the expectation of two languages!
We should design with the expectation of two languages!

Specialize aggressively to simplify the languages
We should design with the expectation of two languages!

Specialize aggressively to simplify the languages.
We should design with the expectation of two languages!

Specialize aggressively to simplify the languages.
We should design with the expectation of two languages!

Specialize aggressively to simplify the languages.
Designing with the expectation of Two Languages

Make each language aware of the other to remove glue.
Designing with the expectation of Two Languages

Make each language aware of the other to remove glue.
Designing with the expectation of Two Languages

Meta-programs

Low

High

Meta-program the low-level language to produce high-performance code from concise descriptions.
Designing with the expectation of Two Languages

Low

Designed to be used with low-level languages such as C
[Jerusalimschy et al. 11]
Designing with the expectation of Two Languages

New low-level language designed to work with high-level languages

Designated to be used with low-level languages such as C

[lerusalimschy et al. 11]
Example: Lua

--this is a comment.
--top level is Lua code:
function add(a,b)
    return a + b
end
print(add(3,4)) --7
Example: Lua + Terra

--this is a comment.
--top level is Lua code:
function add(a,b)
    return a + b
end
print(add(3,4)) --7

--terra introduces a low-level terra function
terra addt(a : int, b : int) : int
    return a + b
end

print(addt(3,4)) --7

Terra function called from Lua
Types and semantics are similar to C

```plaintext
struct FloatArray {
    data: &float;
    N : int;
}

--get an element from the array
terra FloatArray:get(i : int) : float
    return self.data[i]
end
```
Types and semantics are similar to C

```plaintext
struct FloatArray {
    data: &float;
    N : int;
}

--get an element from the array
terra FloatArray:get(i : int) : float
    return self.data[i]
end
```
Types and semantics are similar to C

```plaintext
struct FloatArray {
    data: &float;
    N : int;
}

--get an element from the array
terra FloatArray:get(i : int) : float
    return self.data[i]
end
```
Types and semantics are similar to C

```c
struct FloatArray {
    data: &float;
    N : int;
}

--get an element from the array
terra FloatArray:get(i : int) : float
    return self.data[i]
end
```
3 Design Issues
Design Issue 1: Compartmentalized Runtimes

Terra Runtime

Lua Runtime
Design Issue 1: Compartmentalized Runtimes

Terra Runtime

Lua Runtime

Lua Code

Lua JIT

Lua Bytecode (+ tracing JIT)
Design Issue 1: Compartamentalized Runtimes

Terra Runtime

Lua Runtime

Lua Code

LuaJIT

Lua Heap (GC’d)

Lua Bytecode (+ tracing JIT)
Design Issue 1: Compartmentalized Runtimes

Terra Runtime

- Terra Code
  - LLVM
    - x86 Code
    - GPU Code
    - ARM Code

Lua Runtime

- Lua Code
  - LuaJIT
    - Lua Heap (GC’d)
    - Lua Bytecode (+ tracing JIT)
Design Issue 1: Compartmentalized Runtimes

**Terra Runtime**
- Terra Code
  - LLVM
    - x86 Code
    - GPU Code
    - ARM Code
  - C Heap (Manual)
  - GPU Heap

**Lua Runtime**
- Lua Code
  - LuaJIT
  - Lua Heap (GC’d)
  - Lua Bytecode (+ tracing JIT)
Design Issue 1: Compartmentalized Runtimes

Separation ensures Terra can always produce fast code.
Design Issue 2: Clean interface between languages
Design Issue 2: Clean interface between languages

terra addt(a : int, b : int) : int
    return a + b
end
Design Issue 2: Clean interface between languages

```lua
terra addt(a : int, b : int) : int
    return a + b
end
```

Terra functions are first class Lua values:

```lua
print(addt)
> <terra function>
```
Design Issue 2: Clean interface between languages

```lua
terra addt(a : int, b : int) : int
    return a + b
end

Terra functions are first class Lua values:

```lua
print(addt)
> <terra function>
```

Terra uses Lua’s lexical environment to resolve symbols:

```lua
terra add1(a : int) : int
    return _addt(a,1)
end
```
Design Issue 2: Clean interface between languages

```lua
terra addt(a : int, b : int) : int
    return a + b
end
```

Terra functions are first class Lua values:

```lua
print(addt)
> <terra function>
```

Terra uses Lua’s lexical environment to resolve symbols:

```lua
terra add1(a : int) : int
    return addt(a,1)
end
```

When called from one another, values are translated from one language to another using rules adapted from LuaJIT’s FFI

```lua
print(addt(1,2)) -- on call: lua number -> int
    -- on return: number -> int
```
Design Issue 3: Meta-programming

All Terra entities (types, functions, expressions, symbols) are first-class Lua values.
Design Issue 3: Meta-programming

All Terra entities (types, functions, expressions, symbols) are first-class Lua values.

Ex. Templating:

```lua
local struct ArrayType {
    data : &float;
    N : int;
}
terra ArrayType:get(i: int) : float
    return self.data[i]
end
```
Design Issue 3: Meta-programming

All Terra entities (types, functions, expressions, symbols) are first-class Lua values.

Ex. Templating:

```lua
local struct ArrayType {
  data : &ElemType;
  N : int;
}

terra ArrayType:get(i: int) : ElemType
  return self.data[i]
end
```
Design Issue 3: Meta-programming

All Terra entities (types, functions, expressions, symbols) are first-class Lua values.

Ex. Templating:

```lua
function Array(ElemType)
  local struct ArrayType {
    data : &ElemType;
    N : int;
  }
  terra ArrayType:get(i: int) : ElemType
    return self.data[i]
  end
  return ArrayType
end
FloatArray = Array(float)
```
function gen_square(x)
    return `x * x
end

terra mse(a: float, b: float)
    return [gen_square(a)] - [gen_square(b)]
end

Terra is meta-programmed from Lua using **multi-stage programming** (e.g., from MetaOCaml)
Terra is meta-programmed from Lua using **multi-stage programming** (e.g., from MetaOCaml)

```lua
function gen_square(x)
    return `x * x`
end

terra mse(a: float, b: float)
    return [gen_square(a)] - [gen_square(b)]
end
```

In Lua, a **quotation** creates a Terra expression. Like a “**string literal**” for code.
Terra is meta-programmed from Lua using **multi-stage programming** (e.g., from MetaOCaml)

```lua
function gen_square(x)
    return `x * x`
end
```

In Lua, a **quotation** creates a Terra expression.
Like a “**string literal**” for code.

```terra
terra mse(a: float, b: float)
    return [gen_square(a)] - [gen_square(b)]
end
```

In Terra, an **escape** splices the value of a Lua expression into Terra code.
Like a string interpolation operator “**hello, %s**”
Evaluation Semantics

print("lua execution")

function gen_square(x)
    return `x * x
end

terra sqd(a: float, b: float)
    return [gen_square(a)] - [gen_square(b)]
end

print(mse(3,2))
Evaluation Semantics

print("lua execution")
> lua execution

function gen_square(x)
  return `x * x
end

terra sqd(a: float, b: float)
  return [gen_square(a)] - [gen_square(b)]
end

print(mse(3,2))

1. Lua code evaluates normally until it reaches a Terra function or quote expression
Evaluation Semantics

print("lua execution")
> lua execution

function gen_square(x)
  return `x * x
end

terra sqd(a: float, b: float)
  return [gen_square(a)] - [gen_square(b)]
end

print(mse(3,2))

1. Lua code **evaluates** normally until it reaches a Terra function or quote expression

2. The Terra expression is **specialized**, by evaluating all *escaped* Lua expressions.
print(“lua execution”)

function gen_square(x)
    return `x * x
end

terra sqd(a: float, b: float): float
    return [a * a] - [gen_square(b)]
end

print(mse(3,2))
print("lua execution")

function gen_square(x)
    return x * x
end

terra sqd(a: float, b: float)
    return [ `a * a ] - [ `b * b ]
end

print(mse(3,2))

1. Lua code **evaluates** normally until it reaches a Terra function or quote expression

2. The Terra expression is **specialized**, by evaluating all escaped Lua expressions.
Evaluation Semantics

print("lua execution")

function gen_square(x)
  return `x * x
end

terra sqd(a: float, b: float)
  return a * a - b * b
end

print(mse(3,2))

1. Lua code **evaluates** normally until it reaches a Terra function or quote expression.

2. The Terra expression is **specialized**, by evaluating all *escaped* Lua expressions.
Evaluation Semantics

print("lua execution")

function gen_square(x)
    return `x * x`
end

terra sqd(a: float, b: float)
    return a * a - b * b
end

print(mse(3,2))
Evaluation Semantics

print("lua execution")

function gen_square(x)
    return `x * x`
end

terra sqd(a: float, b: float)
    return a * a - b * b
end

print(mse(3,2))
> 5

1. Lua code evaluates normally until it reaches a Terra function or quote expression.

2. The Terra expression is specialized, by evaluating all escaped Lua expressions.

3. The Terra function is evaluated as Terra.
Programmatically decide memory layout of types

```
terra example()
    var s : Student
    s:setname("bob")
    s:setyear(4)
end
```
Programmatically decide memory layout of types

**Example**

```terra
example()

var s : Student
s:setname("bob")

end
```

Like a high-level language: generate types using dynamic information.
Programmatically decide memory layout of types

Like a high-level language: generate types using dynamic information.

define example()
  var s : Student
  s:setname("bob")
  s:setyear(4)
end

Studentmetatables.__getentries = function()
  file = io.open("students.dat","r")
  return entries
end

create ORM by populating type with fields described in database file
Programmatically decide memory layout of types

```terra
example()

var s : Student
s:setname("bob")
s:setyear(4)
end
```

Student.metamethods.__getentries = function()
  file = io.open("students.dat","r")
  return entries
end

create ORM by populating type with fields described in database file

Like a high-level language: generate types using dynamic information.
Like a low-level language: optimize code using memory layout.

use generated layout in compiler optimizations
Programmatically decide memory layout of types

Object behavior can also be meta-programmed.

Like a high-level language: generate types using dynamic information.
Like a low-level language: optimize code using memory layout.
Experiences Using Terra
Hypothesis  We can implement programs that generate code using Lua-Terra that are *concise* and *fast* compared to the state-of-the-art.
Auto-tuning: Matrix Multiply

Highly tuned and sensitive to architectural differences.

Compare to: ATLAS an auto-tuner for BLAS

Implemented ATLAS’s approach in Terra

- vectorization, e.g. `vector(float,4)`
- register blocking
- unrolling
- cache blocking
Terra
Matrix Multiply
Architecture

Search space of kernel parameters

codegen, using multi-stage operators

Generated inner kernel code

Wrapper API
Terra
Matrix Multiply
Architecture

Search space of kernel parameters

Lua

codegen, using multi-stage operators

Generated inner kernel code

Terra

Wrapper API

Terra

Perf. evaluation, using runtime code generation and timing

\[
\begin{bmatrix}
0 & 1 & 2 & 0 \\
3 & 4 & 0 & 0 \\
5 & 0 & 0 & 0 \\
0 & 0 & 6 & 7 \\
\end{bmatrix}
\begin{bmatrix}
1 \\
4 \\
2 \\
3 \\
\end{bmatrix}
= 
\begin{bmatrix}
8 \\
19 \\
5 \\
33 \\
\end{bmatrix}
\]
Terra
Matrix Multiply
Architecture

Search space of kernel parameters

Lua

~250 lines

Generated inner kernel code

Terra

Perf. evaluation, using runtime code generation and timing

Wrapper API

Terra

\[
\begin{bmatrix}
0 & 1 & 2 & 0 \\
3 & 4 & 0 & 0 \\
5 & 0 & 0 & 0 \\
0 & 0 & 6 & 7
\end{bmatrix}
\begin{bmatrix}
1 \\
4 \\
2 \\
3
\end{bmatrix}
= 
\begin{bmatrix}
8 \\
19 \\
5 \\
33
\end{bmatrix}
\]
Matrix Multiply, performance results

double precision

Within 20%, but ATLAS is auto-tuning x86 ASM directly!

Run on an Intel Core i7-3720QM
Matrix Multiply, performance results

Competitive with the MKL, within 60% of peak theoretical FLOPS!

Run on an Intel Core i7-3720QM
Terra-Lua simplifies architecture

ATLAS auto-tuner thousands of lines of C/C, Makefiles, templating language

Terra auto-tuner 250 of lines of Terra/Lua
ATLAS needed custom tools for each task

- Search space of kernel parameters
  - Custom Templating, C++ programs, Makefiles
  - codegen, source-to-source, C-preprocessor
  - Generated inner kernel code
    - x86 ASM (and other ASMs)
  - Wrapper API
    - C/C++

Perf. evaluation, offline process using Makefiles
Dynamic x86 Assembly
Dynamic x86 Assembly

Want to dynamically generate x86 assembly. For example, Riposte [Talbot et al. 12] (a JIT compiler for R) uses code like this:

```c
void emitGather(Assembler &a,
    XMMRegister RegR,
    XMMRegister RegA,
    int disp) {
    a.movq(r8, RegA);
    a.movhlps(RegR, RegA);
    a.movq(r9, RegR);
    a.movlpd(RegR,Operand(r12,r8,times_8,disp));
    a.movhpd(RegR,Operand(r12,r9,times_8,disp));
}
```
Dynamic x86 Assembly

Want to dynamically generate x86 assembly. For example, Riposte [Talbot et al. 12] (a JIT compiler for R) uses code like this:

```c
void emitGather(Assembler &a,
    XMMRegister RegR,
    XMMRegister RegA,
    int disp) {
    a.movq(r8, RegA);
    a.movhlps(RegR, RegA);
    a.movq(r9, RegR);
    a.movlpsd(RegR,Operand(r12,r8,times_8,disp));
    a.movhpsd(RegR,Operand(r12,r9,times_8,disp));
}
```
Dynamic x86 Assembly

Want to dynamically generate x86 assembly. For example, Riposte [Talbot et al. 12] (a JIT compiler for R) uses code like this:

```c
void emitGather(Assembler &a,
                XMMRegister RegR,
                XMMRegister RegA,
                int disp) {
  a.movq(r8, RegA);
  a.movhlps(RegR, RegA);
  a.movq(r9, RegR);
  a.movlpd(RegR, Operand(r12,r8,times_8,disp));
  a.movhpd(RegR, Operand(r12,r9,times_8,disp));
}
```

Problem: there are hundreds of x86 instructions, each a different method.
We can use a small language to describe instructions

The DynAsm used in LuaJIT describes instructions in a small language:

```
mov1lpd_2  = "rx/oq:660F12rM|xr/qo:n660F13Rm"
```

336 KB/s when interpreted directly, 168 MB/s using DynAsm’s source-to-source translation that optimizes code size
We can use a small language to describe instructions

The DynAsm used in LuaJIT describes instructions in a small language:

\[
\text{movlpd}_2 = "rx/oq:660F12rM|xr/qo:n660F13Rm"
\]

336\text{KB/s} when interpreted directly, 168\text{MB/s} using DynAsm’s source-to-source translation that optimizes code size

We can use Terra to generate our assembler type based on this language!
**A Terra Implementation based on DynASM**

```plaintext
terra emitGather(RegR : O, RegA : O, disp : int)
  A:emit(op.movd, r8, RegA,
  op.movhlps, RegR, RegA,
  op.movd, r9, RegR,
  op.movlpd, RegR, index(r12, r8, 8, disp, "qword"),
  op.movhpd, RegR, index(r12, r9, 8, disp, "qword"))
end
```
A Terra Implementation based on DynASM

terra emitGather(RegR : O, RegA : O, disp : int)
  A:emit(op.movd, r8, RegA,
    op.movh1ps, RegR, RegA,  
    op.movd, r9, RegR,
    op.movlpd, RegR, index(r12, r8, 8, disp, "qword"),
    op.movhpd, RegR, index(r12, r9, 8, disp, "qword"))
end

Dynamically determined
A Terra Implementation based on DynASM

terra emitGather(RegR : O, RegA : O, disp : int)
  A:emit(op.movd, r8, RegA,
       op.movhlps, RegR, RegA,
       op.movd, r9, RegR,
       op.movlpd, RegR, index(r12, r8, 8, disp, "qword"),
       op.movhpd, RegR, index(r12, r9, 8, disp, "qword"))
end
A Terra Implementation based on DynASM

```
terra emitGather(RegR : 0, RegA : 0, disp : int)
  A:emit(op.movd, r8, RegA, op.movh1ps, RegR, RegA, op.movd, r9, RegR, op.movl1pd, RegR, index(r12, r8, 8, disp, "qword"), op.movh1pd, RegR, index(r12, r9, 8, disp, "qword"))
end
```

Dynamically determined:

Static:

At typechecking, use the Assembly type to generates a template:

```
10001XXX10101XXX1010XXX010101000XXX0101010XXXXXXXX10101010XXXXXXXX
```
A Terra Implementation based on DynASM

```terra
emitGather(RegR : 0, RegA : 0, disp : int)
  A:emit(op.movd, r8, RegA,
         op.movhlp, RegR, RegA,
         op.movd, r9, RegR,
         op.movlpg, RegR, index(r12, r8, 8, disp, "qword"),
         op.movhpd, RegR, index(r12, r9, 8, disp, "qword"))
end
```

At typechecking, use the Assembly type to generates a template:

```
10001XXX10101XXX1010XXX01010100XXX0101010XXXXXXXX10101010XXXXXXXX
```

At runtime:
1. Copy template into code (vectorized! 16-bytes at a time)
2. Patch template with dynamic variables
A Terra Implementation based on DynASM

```terra
terra emitGather(RegR : 0, RegA : 0, disp : int)
  A:emit(op.movd, r8, RegA,
    op.movhlp, RegR, RegA,
    op.movd, r9, RegR,
    op.movlpd, RegR, index(r12, r8, 8, disp, "qword"),
    op.movhpd, RegR, index(r12, r9, 8, disp, "qword"))
end
```

Dynamically determined

Static

At typechecking, use the Assembly type to generates a template:

```
10001XXX1010101XXX1010XXX010101000XXX0101010XXXXXXXXX10101010XXXXXXXXX
```

At runtime:
1. Copy template into code (vectorized! 16-bytes at a time)
2. Patch template with dynamic variables
Terra ASM is 3 – 20 times faster than Chrome's.
Terra ASM 1/2 the size of Google's ASM

```cpp
void Assembler::movlpd(XMMRegister dst, const Operand& src)
    {
    EnsureSpace ensure_space(this);
    emit(0x66);
    emit_rex_64(dst, src);
    emit(0x0F);
    emit(0x12); // load
    emit_sse_operand(dst, src);
    }
```

```
movlpd_2 = "rx/oq:660F12rM|xr/qo:n660F13Rm"
```

VS.

```cpp
void Assembler::movlpd(XMMRegister dst, const Operand& src) {
    EnsureSpace ensure_space(this);
    emit(0x66);
    emit_rex_64(dst, src);
    emit(0x0F);
    emit(0x12); // load
    emit_sse_operand(dst, src);
}
```
Experiences Building DSLs
DSLs: Darkroom, a stencil language for image processing

Similar in concept to the Halide language

[J. Ragan-Kelly et al. 12]
DSLs are embedded in Lua, just like Terra

```lua
bx = im(x, y)
   (I(x-1,y) + I(x,y) + I(x+1,y))/3
end
```
DSLs are embedded in Lua, just like Terra

\[ bx = \text{im}(x, y) \]

\[
(I(x-1,y) + I(x,y) + I(x+1,y))/3
\]

Start token transfers control from Lua to user-defined parser
DSLs are embedded in Lua, just like Terra

Start token transfers control from Lua to user-defined parser

\[
\text{bx} = \text{im}(x, y) \\
(I(x-1,y) + I(x,y) + I(x+1,y))/3
\]

end

DSLs \textbf{also} benefit from meta-programming.
Darkroom Architecture

- **Frontend**
  - Fusion, vectorization, line buffering
  - Lua, via language extension
  - codegen, using staging
  - Terra
  - Generated Kernel code
  - Terra
- **Wrapper loads images and launches threads**
  - Terra
Darkroom, speed-up over C

Fluid Simulation:

Reference C  1x (37 sec)
Matching Orion  1x (37 sec)
+ Vectorization  1.9x (20 sec)
+ Line buffering  2.3x (16 sec)

Separated Area Filter:

Reference C  1x (4.4 ms)
Matching Orion  1.1x (4.1 ms)
+ Vectorization  2.8x (1.6 ms)
+ Line Buffering  3.4x (1.3 ms)
Darkroom has comparable performance to Halide

<table>
<thead>
<tr>
<th></th>
<th>Orion</th>
<th>Halide</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deblur</td>
<td>43 MP/s</td>
<td>46 MP/s</td>
</tr>
</tbody>
</table>

[Hegarty et al. 14]
Halide 1.0 Architecture

- Frontend
- Fusion, vectorization, line buffering
- Generated Kernel code
- Wrapper loads images and launches threads

C++ operators overloading
OCaml
codegen, using LLVM
LLVM
C++

53
Halide 1.0
Architecture

Frontend

C++ operators overloading

Fusion, vectorization, line buffering

OCaml

codegen, using LLVM

Generated Kernel code

LLVM

Wrapper loads images and launches threads

C++
Building High-performance languages using Terra

**Darkroom**  Language for image processing using stencils. Has comparable CPU performance to the LLVM-based Halide DSL. ([http://darkroom-lang.org](http://darkroom-lang.org), [Hegarty et al. SIGGRAPH 14])

**Quicksand**  Language for probabilistic computing. Order-of-magnitude faster than probabilistic languages written in high-level languages. ([http://dritchie.github.io/quicksand](http://dritchie.github.io/quicksand), [Ritchie et al. SIGGRAPH 15])

**Liszt/Ebb**  Language for physical simulations on GPUs ([http://arxiv.org/abs/1506.07577](http://arxiv.org/abs/1506.07577), [Bernstein et al., in sub. to TOG])

Take Away  A two language approach makes it significantly simpler to architect libraries and DSLs that need to generate high-performance code.
Using compilers like LLVM to dynamically generate code tedious and verbose

```c
float solve(float a, float b, float c) {
    return (-b + sqrt(b*b - 4*a*c)) / (2 * a);
}
```
Using compilers like LLVM to dynamically generate code tedious and verbose

```c
float solve(float a, float b, float c) {
    return (-b + sqrt(b*b - 4*a*c)) / (2 * a);
}
```

```c
Value* float_mul = B.CreateFMul(float_b, float_b);
Value* float_mul1 = B.CreateFMul(float_a, const_float_3);
Value* float_mul2 = B.CreateFMul(float_mul1, float_c);
Value* float_sub3 = B.CreateFSub(float_mul, float_mul2);
Value* float_call = B.CreateCall(func_sqrtf, float_sub3);
Value* float_add = B.CreateFSub(float_call, float_b);
Value* float_div = B.CreateFMul(float_add, const_float_4);
Value* float_mul4 = B.CreateFMul(float_div, float_a);
B.CreateReturn(float_mul4);
```
Using compilers like LLVM to dynamically generate code tedious and verbose.

```cpp
float solve(float a, float b, float c) {
  return (-b + sqrt(b*b - 4*a*c)) / (2 * a);
}

std::vector<Type*> SolveTy_args;
SolveTy_args.push_back(Type::getFloatTy(C));
SolveTy_args.push_back(Type::getFloatTy(C));
SolveTy_args.push_back(Type::getFloatTy(C));
FunctionType* SolveTy = FunctionType::get(Type::getFloatTy(C), SolveTy_args);

std::vector<Type*> SqrtTy_args;
SqrtTy_args.push_back(Type::getFloatTy(C));
FunctionType* SqrtTy = FunctionType::get(Type::getFloatTy(C), SqrtTy_args);

PointerType* PtrSqrtTy =PointerType::get(SqrtTy, 0);

Function* func_solve = Function::Create(SolveTy,
  GlobalValue::ExternalLinkage,
  "solve", M);
Function* func_sqrtf = Function::Create(SqrtTy,
  GlobalValue::ExternalLinkage,
  "sqrtf", M);

ConstantFP* const_float_3 = ConstantFP::get(C, 4.f);
ConstantFP* const_float_4 = ConstantFP::get(C, 5.f);

Function::arg_iterator args = func_solve->arg_begin();
Value* float_a = args++;
Value* float_b = args++;
Value* float_c = args++;

BasicBlock* label_entry = BasicBlock::Create(C, "entry", func_solve, 0);
IRBuilder<> * B(label_entry);
Value* float_mul = B.CreateFMul(float_b, float_b);
Value* float_mul1 = B.CreateFMul(float_a, const_float_3);
Value* float_mul2 = B.CreateFMul(float_mul1, float_c);
Value* float_sub3 = B.CreateFSub(float_mul, float_mul2);
Value* float_call = B.CreateCall(func_sqrtf, float_sub3);
Value* float_add = B.CreateFAdd(float_call, float_b);
Value* float_div = B.CreateFMul(float_add, const_float_4);
Value* float_mul4 = B.CreateFMul(float_div, float_a);
B.CreateReturn(float_mul4);
```
Using compilers like LLVM to dynamically generate code tedious and verbose.

```cpp
using compilers like LLVM to dynamically generate code tedious and verbose.

//types
std::vector<Type*> SolveTy_args;
SolveTy_args.push_back(Type::getFloatTy(C));
SolveTy_args.push_back(Type::getFloatTy(C));
SolveTy_args.push_back(Type::getFloatTy(C));
FunctionType* SolveTy = FunctionType::get(Type::getFloatTy(C), SolveTy_args);

std::vector<Type*> SqrtTy_args;
SqrtTy_args.push_back(Type::getFloatTy(C));
FunctionType* SqrtTy = FunctionType::get(Type::getFloatTy(C), SqrtTy_args);

PointerType* PtrSqrtTy = PointerType::get(SqrtTy, 0);

//function declarations
Function* func_solve = Function::Create(SolveTy, 
  GlobalValue::ExternalLinkage, 
  "solve", M);
Function* func_sqrtf = Function::Create(SqrtTy, 
  GlobalValue::ExternalLinkage, 
  "sqrtf", M);

// constants
ConstantFP* const_float_3 = ConstantFP::get(C, 4.f);
ConstantFP* const_float_4 = ConstantFP::get(C, 5.f);

// function definition
Function::arg_iterator args = func_solve->arg_begin();
Value* float_a = args++;
Value* float_b = args++;
Value* float_c = args++;
BasicBlock* label_entry = BasicBlock::Create(C, "entry", func_solve, 0);
IRBuilder<> * B(label_entry);
Value* float_mul = B.CreateFMul(float_b, float_b);
Value* float_mul1 = B.CreateFMul(float_a, const_float_3);
Value* float_mul2 = B.CreateFMul(float_mul1, float_c);
Value* float_sub3 = B.CreateFSub(float_mul, float_mul2);
Value* float_call = B.CreateCall(func_sqrtf, float_sub3);
Value* float_div = B.CreateFSub(float_call, float_b);
Value* float_mul4 = B.CreateFMul(float_div, float_a);
B.CreateReturn(float_mul4);

//function solve(a, b, c)
return -(-b + sqrt(b^2 - 4*a*c))/(2*a)
end
```
Serialization

Automatically generate code to serialize object trees.
1. introspect user-defined types to generate default serialization behavior
Serialization

Automatically generate code to serialize object trees.
1. introspect user-defined types to generate default serialization behavior
Serialization

Automatically generate code to serialize object trees.
1. introspect user-defined types or generate default serialization behavior
Serialization

Automatically generate code to serialize object trees.
1. introspect user-defined types or generate default serialization behavior
2. generate aggressive optimized code based on circumstances.
Serialization

Automatically generate code to serialize object trees.

1. introspect user-defined types to generate default serialization behavior

2. generate aggressive optimized code based on circumstances.
   (a) specialize write function
Serialization

Automatically generate code to serialize object trees.

1. introspect user-defined types to generate default serialization behavior

2. generate aggressive optimized code based on circumstances.
   (a) specialize write function
   (b) fuse serialization across objects to generate bigger memcpy calls

![Graph showing Encoding Throughput (MB/s) for Java, Kryo, and Terra]
Serialization

Automatically generate code to serialize object trees.
1. introspect user-defined types o generate default serialization behavior
2. generate aggressive optimized code based on circumstances.
   (a) specialize write function
   (b) fuse serialization across objects to generate bigger memcpy calls
Serialization

Automatically generate code to serialize object trees.

1. introspect user-defined types to generate default serialization behavior

2. generate aggressive optimized code based on circumstances.
   (a) specialize write
   (b) fuse serialization across objects to generate bigger memcpy calls

---

[Bar chart showing encoding throughput (MB/s) for Java, Kryo, C++ Google protobuf, and Terra.]

- Java: ~200 lines
- Kryo: ~200 lines
- C++ Google protobuf: ~200 lines
- Terra: ~200 lines

Specialize write
Fusion