All the OpenCL on GitHub

Teaching an AI to code, one character at a time.

http://chriscummins.cc

2013 


2014 

2015 

2016 

+


Validating *portability* of optimisations is really hard.

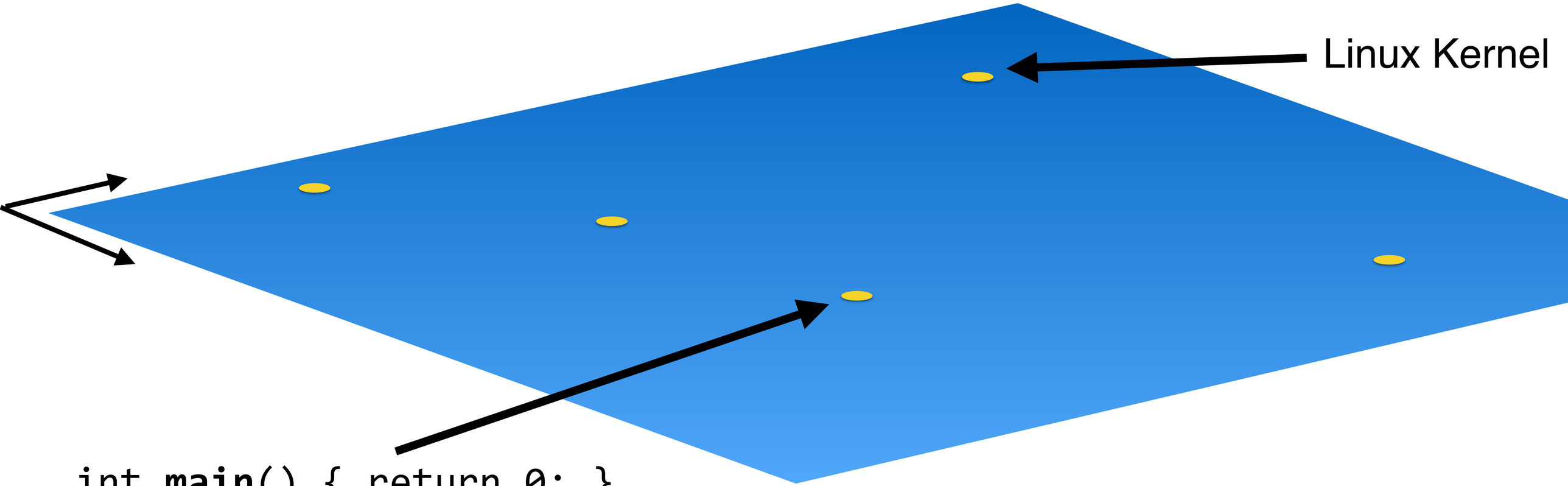
Best practise:

Use benchmark suite of 10-20 programs.
e.g. Rodinia, CompuBench.
Cross-validate results to “prove” portability.

Is this enough?

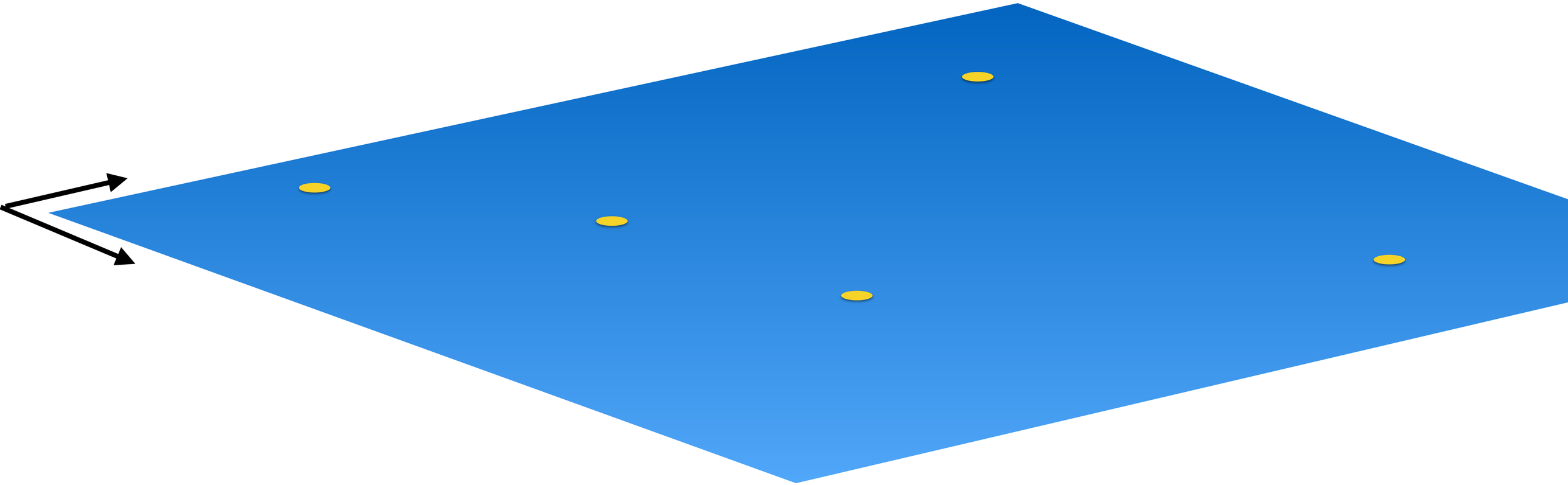
Consider the space of program implementations.

10-20 programs is a *very* small sample size.



Leads to over-fitting optimisations to benchmarks.

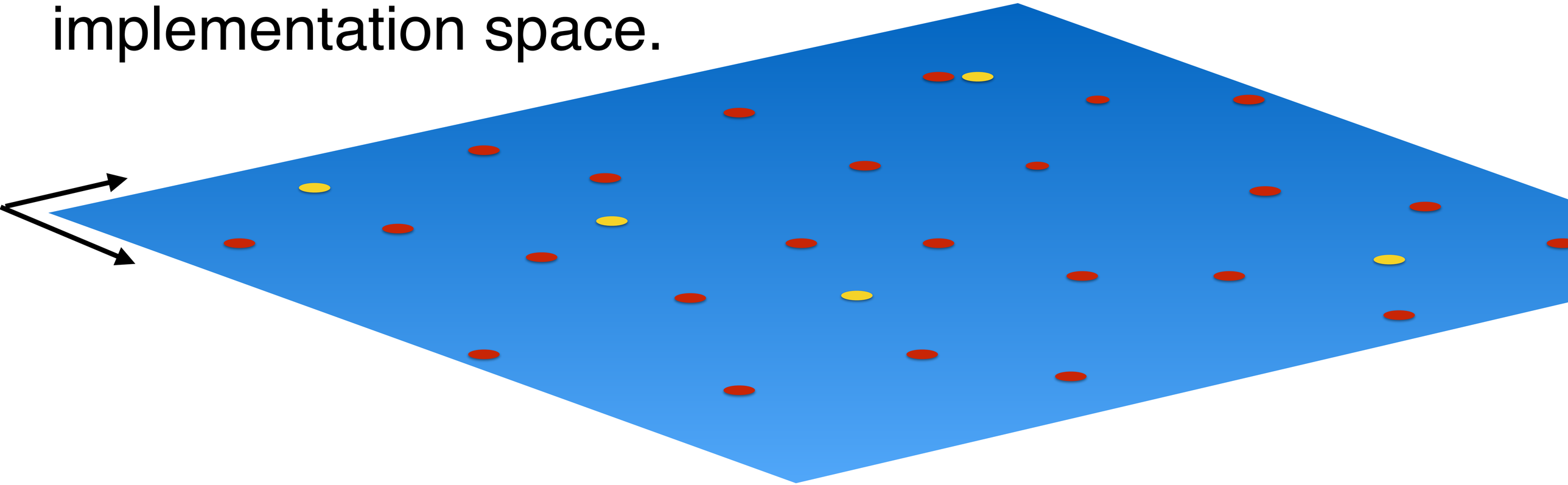
(this is an open secret)



Can anything be done about this?

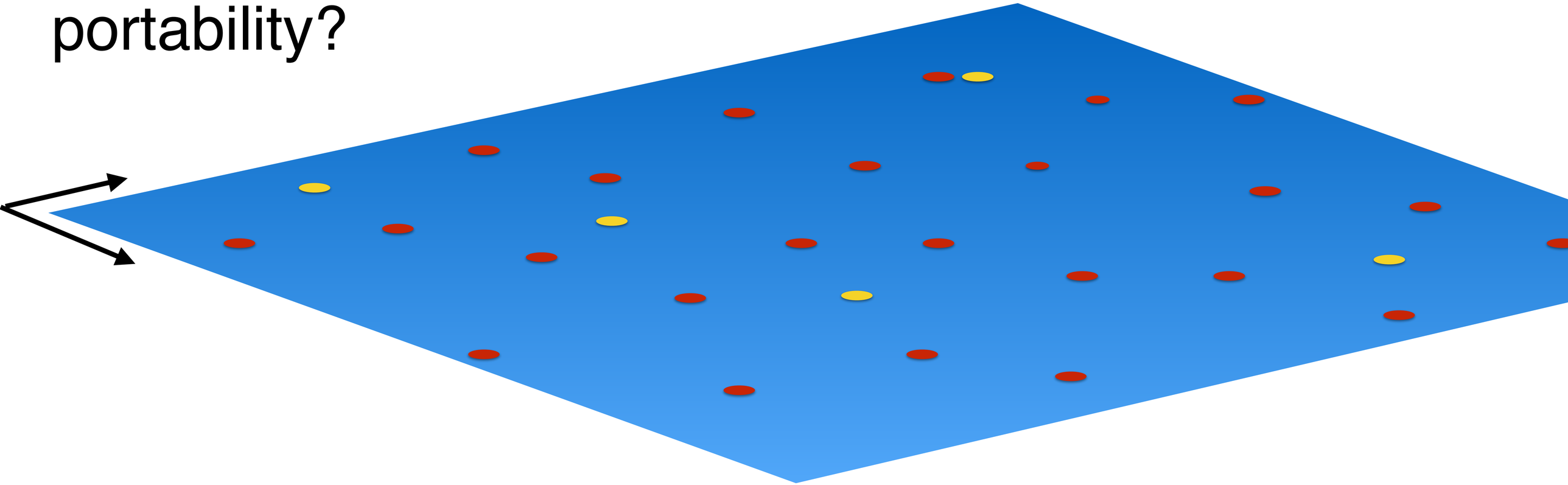
Compiler fuzzing is a promising technique for testing.

Provides uniform sampling across (constrained) implementation space.

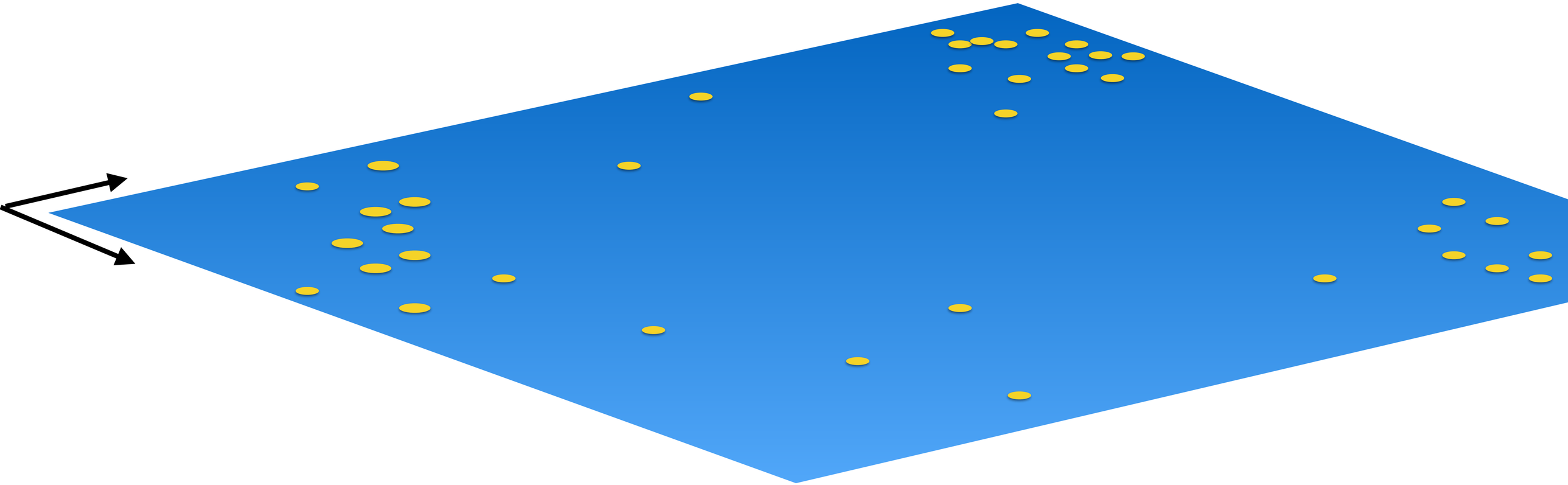


Great for testing **functional** portability of compilers.

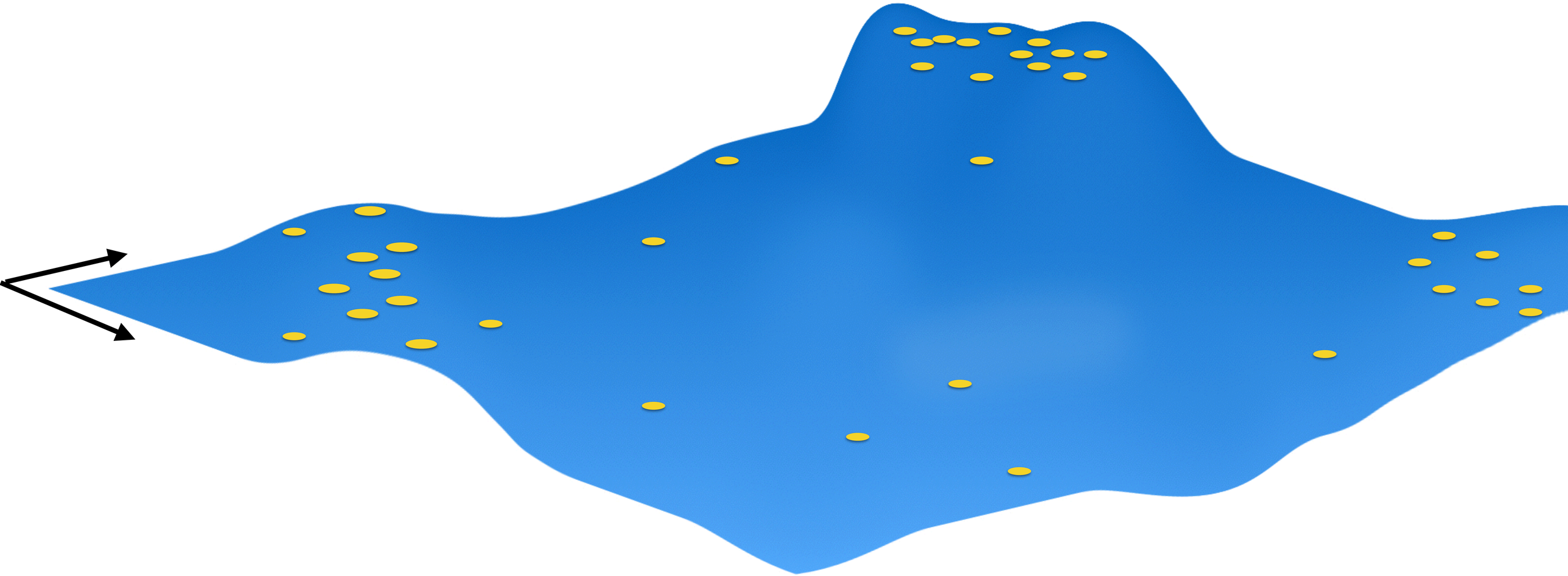
Can we use something similar for **performance** portability?



Hypothesis: Real source codes form *clusters* in the implementation space.

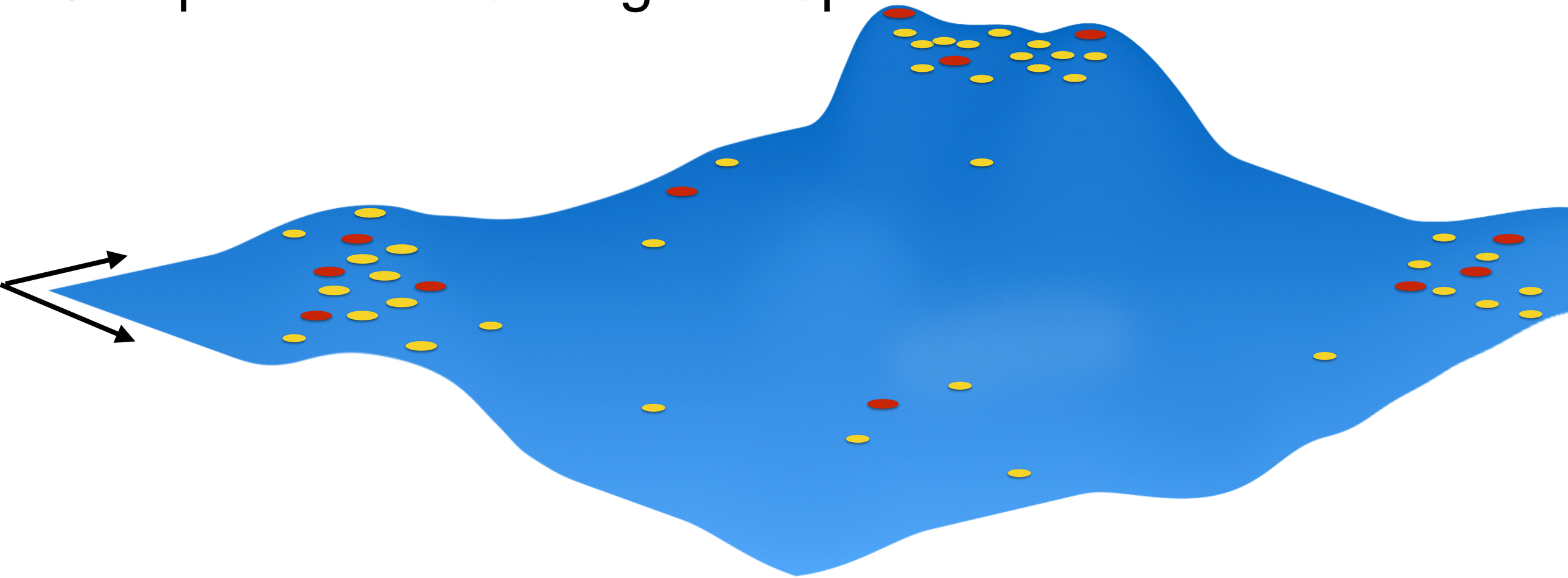


Weight the program space to match the clustering.



Weight the program space to match the clustering.

Sample from this weighted space.



The idea 

Build a probabilistic representation of the *implementation space* of OpenCL.

Non-uniformly sample from this space to generate synthetic, but *representative* benchmarks.

Methodology

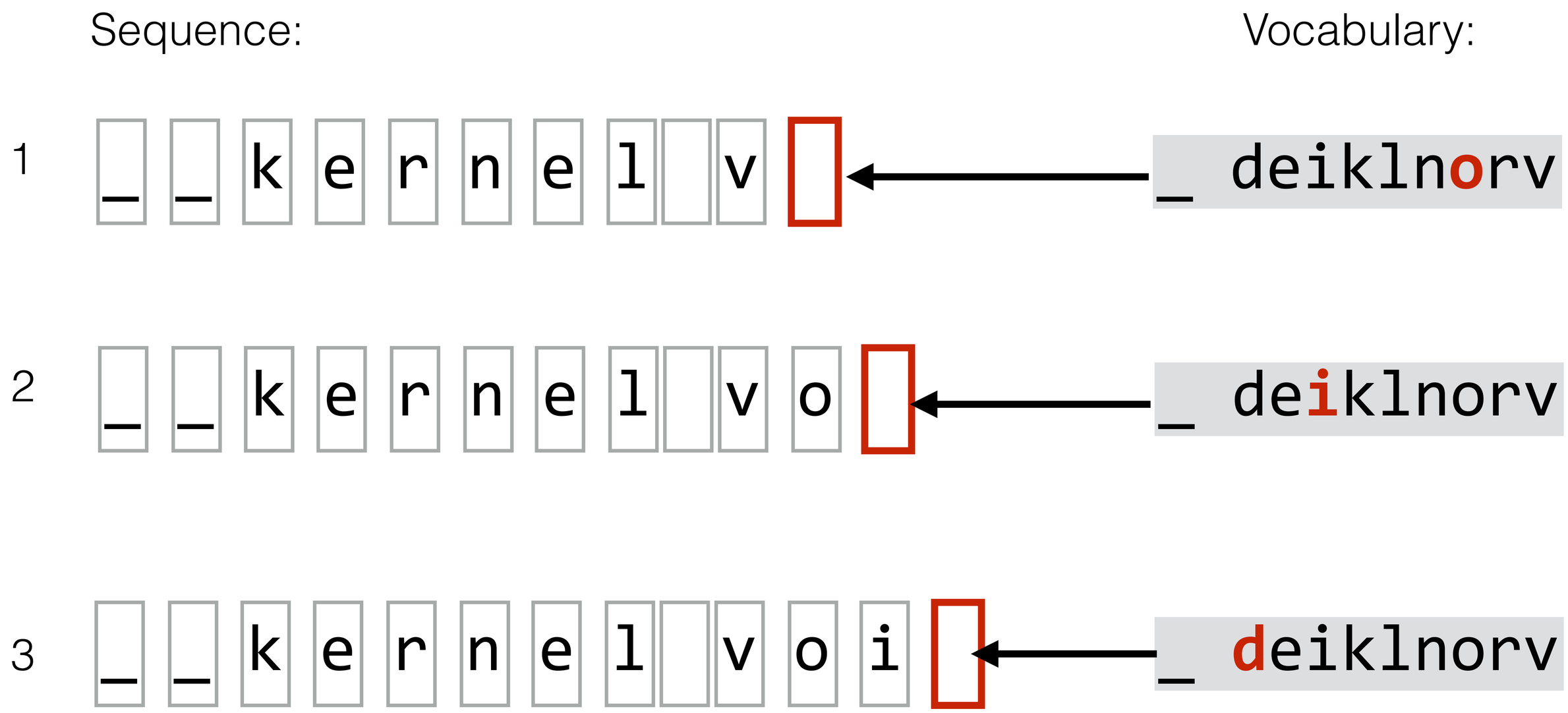
(first steps)

Assemble representative codes in a *language corpus*.

Use a neural network (LSTM) to learn the *character-level* distribution of the language.

Use this distribution to guide the creation of new programs.

Character-level language models:



Character-level language models:

Very low level representation.

LSTMs can learn complex structures over long sequences of events.

Requires a large dataset.

We need to gather a lot of real source codes.

Huge repository of public knowledge: GitHub

(> 35 million repos)

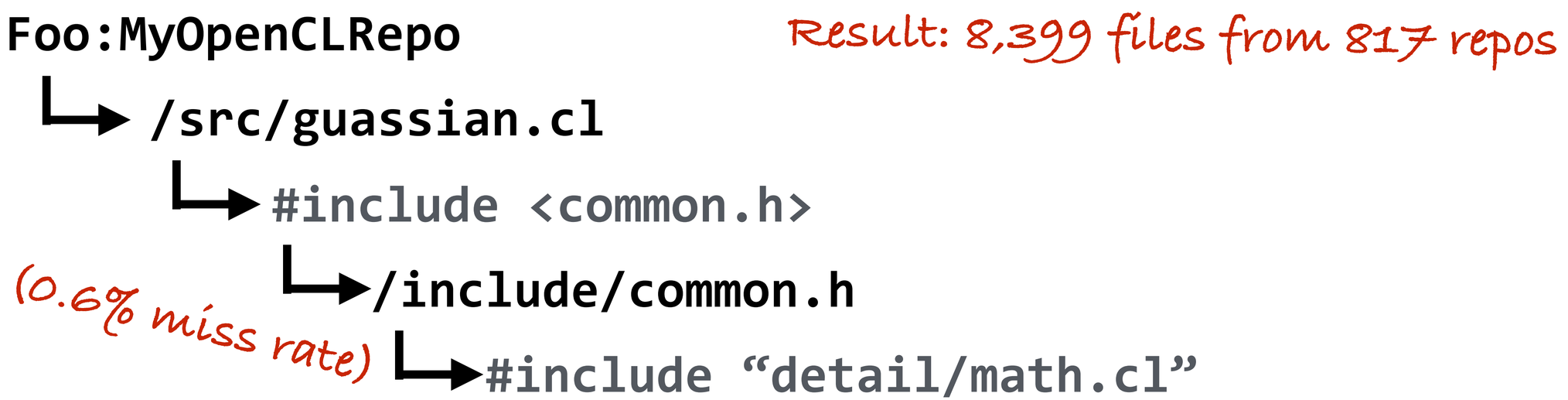
And they have an API :-)

```
$ curl https://api.github.com/search/repositories\?q\=opencl
\&sort\=stars\&order\=desc
{
  "total_count": 3155,
  "incomplete_results": false,
  "items": [
    {
      "id": 7296244,
      "name": "lwjgl3",
      "full_name": "LWJGL/lwjgl3",
```

OpenCL is not a first-class language.

Search repositories using loose keyword terms.
e.g. “`opencl`”, “`nvidia`”, “`gpu`”, “`cl`”, “`amd`”.

Recursively iterate over git trees to get **`.cl`** files.



```
/* Copyright (C) 2004 Joe Bloggs <joe@bloggs.io>  */
//
// Everyone is permitted to copy and distribute verbatim or modified
// copies of this license document, and changing it is allowed as long
// as the name is changed.
//
#define CLAMPING
#define THRESHOLD_MIN 1.0f
#define THRESHOLD_MAX 1.0f

float myclamp(float in) {
#ifdef CLAMPING
    return in > THRESHOLD_MAX ? THRESHOLD_MAX : in < THRESHOLD_MIN ? THRESHOLD_MIN : in;
#else
    return in;
#endif // CLAMPING
}

// Do something really flipping cool
__kernel void findAllNodesMergedAabb(__global float* in, __global float* out, int num_elems)
{
    //
    //
    int id = get_global_id(0);
    if (id < num_elems)
    {


        out[id] = myclamp(in[id]);
    }
}
```

Example file

Is this real, valid OpenCL?

Can we minimise non-functional variance?

Strip comments

```
#define CLAMPING
#define THRESHOLD_MIN 1.0f
#define THRESHOLD_MAX 1.0f

float myclamp(float in) {
#ifdef CLAMPING
    return in > THRESHOLD_MAX ? THRESHOLD_MAX : in < THRESHOLD_MIN ? THRESHOLD_MIN : in;
#else
    return in;
#endif
}

__kernel void findAllNodesMergedAabb(__global float* in, __global float* out, int num_elems)
{
    int id = get_global_id(0);
    if (id < num_elems)
    {


        out[id] = myclamp(in[id]);
    }
}
```

~~Strip comments~~

Preprocess

```
float myclamp(float in) {
   return in > 1.0f ? 1.0f : in < 0.0f ? 0.0f : in;
}

__kernel void findAllNodesMergedAabb(__global float* in, __global float* out, int num_elems)
{
    int id = get_global_id(0);
    if (id < num_elems)
    {


        out[id] = myclamp(in[id]);
    }
}
```

Does it compile?

Does it contain instructions?

~~Strip comments~~

~~Preprocess~~

Rewrite function names

```
float A(float in) {
   return in > 1.0f ? 1.0f : in < 0.0f ? 0.0f : in;
}

__kernel void B(__global float* in, __global float* out, int num_elems)
{
    int id = get_global_id(0);
    if (id < num_elems)
    {


        out[id] = A(in[id]);
    }
}
```

~~Does it compile?~~

~~Does it contain instructions?~~

```
float A(float a) {
   return a > 1.0f ? 1.0f : a < 0.0f ? 0.0f : a;
}

__kernel void B(__global float* a, __global float* b, int c)
{
    int d = get_global_id(0);
    if (d < c)
    {


        b[d] = A(a[d]);
    }
}
```

~~Strip comments~~

~~Preprocess~~

~~Rewrite function names~~

Rewrite variable names

~~Does it compile?~~

~~Does it contain instructions?~~

```
float A(float a) {
  return a > 1.0f ? 1.0f : in < 0.0f ? 0.0f : a;
}

__kernel void B(__global float* a, __global float* b, int c) {
  int d = get_global_id(0);
  if (d < c) {
    b[d] = A(a[d]);
  }
}
```

~~Strip comments~~

~~Preprocess~~

~~Rewrite function names~~

~~Rewrite variable names~~

Enforce code style

~~Does it compile?~~

~~Does it contain instructions?~~

Result: Language corpus (*aka* all the  on ):

4,920 files of preprocessed OpenCL (59%).
1,283,099 lines of code (**30M** characters).

Train an LSTM network on this corpus:

1024 nodes, 3 layers, 1x GPU for ~3 days.

What if we sample the learned distribution directly?

Extract kernels from output stream.

Feed back through preprocessing pipeline.

What if we sample the learned distribution directly?

```
__kernel void A(__global float* a, __global float* b, __global float* c) {
  int d = get_global_id(0);
  if (d >= a) {
    return;
  }
  b[2 * d + 1] = 0.f;

  a[get_global_id(0)] = get_global_id(0);

  barrier(1);

  b[get_global_id(0)] =
      dot(b[get_global_id(0)] * c[get_global_id(2) + get_global_id(0)], 0);
}
```

What if we sample the learned distribution directly?

```
__kernel void A(__global float* a, __local float* b) {
  int c = get_global_id(0);
  if (c < b) {
    a[c] = 1;
    for (int d = 0; d < 16; d++) {
      a[d] = 0;
    }
  }
}

__kernel void B(__global const float *a, __local float *b, __global float *c) {
  int d = get_local_id(0);

  c[d] = a[d] + b[d];
}
```

What if we sample the learned distribution directly?

```
__kernel void A(__global int* a, __global int* b, int c, int d) {
  int e = get_global_id(0);
  if (e >= d) return;

  int f = e / 2 + 1;
  int g = (d - 1) / 2;
  int h = (e + 1) / 2;

  int i;
  int j = 0;
  int k = 0;

  switch (j) {
    case 0:
      if (i <= g) k = j * d + e;
  }
}
```

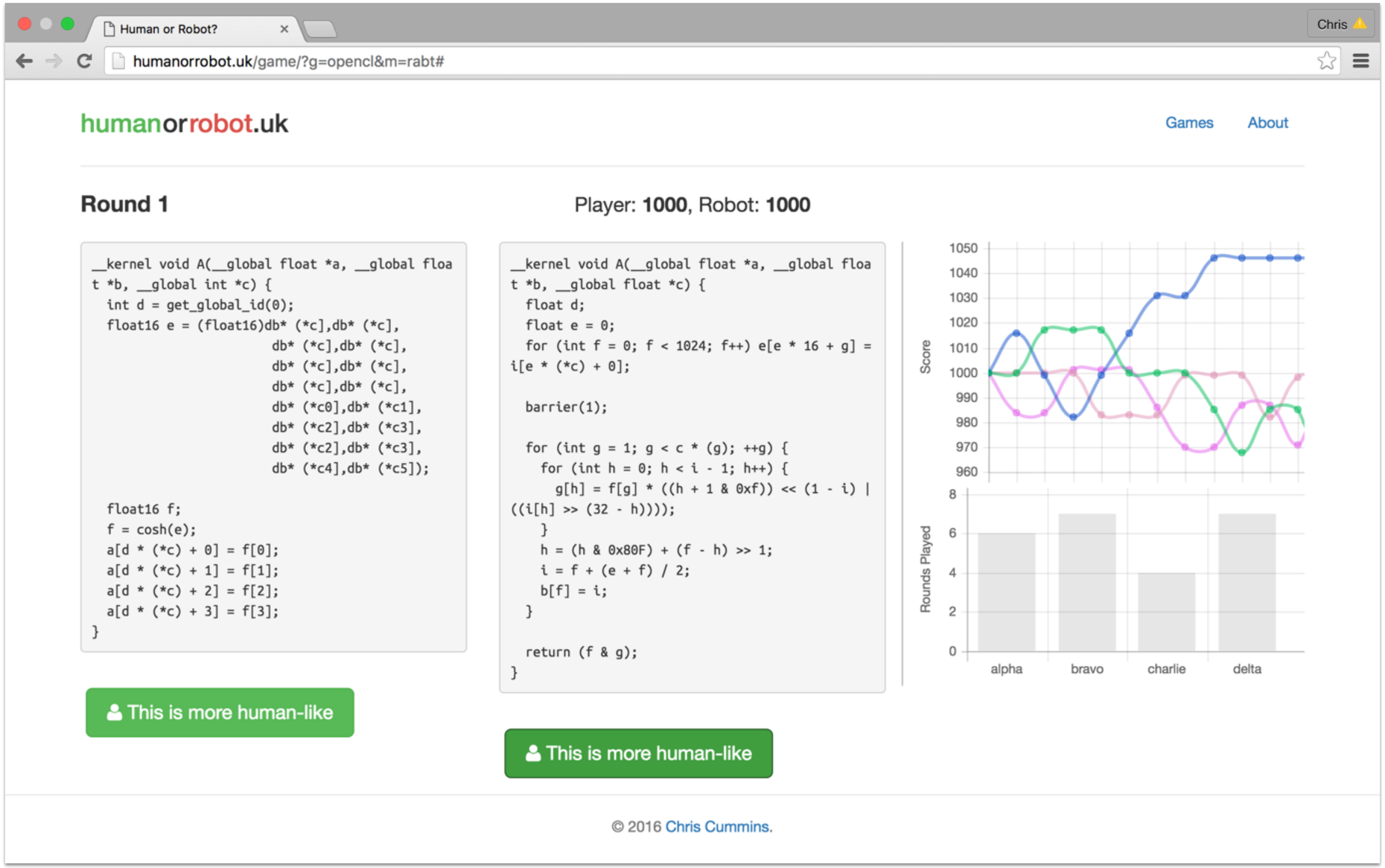
What if we sample the learned distribution directly?

```
__kernel void A(__global float *a) { *a *= -1; }

__kernel void A(__local float *a, int b) {
  const int c = get_global_id(0);
  const int d = get_global_id(1);
  if (d >= a) {
    return;
  }
  const int e = get_global_id(1);
  if (e < d) {
    a[d] = e;
  }
}

__kernel void A(__local float *a) { a[get_global_id(0)] = get_local_id(0); }
```

How do we evaluate machines at human-like tasks?



Next steps:

Higher-level language model (e.g. token stream).

Static analysis OR embed DNN in fuzzing logic.

Crowdsourced Turing Tests to guide DNN design.

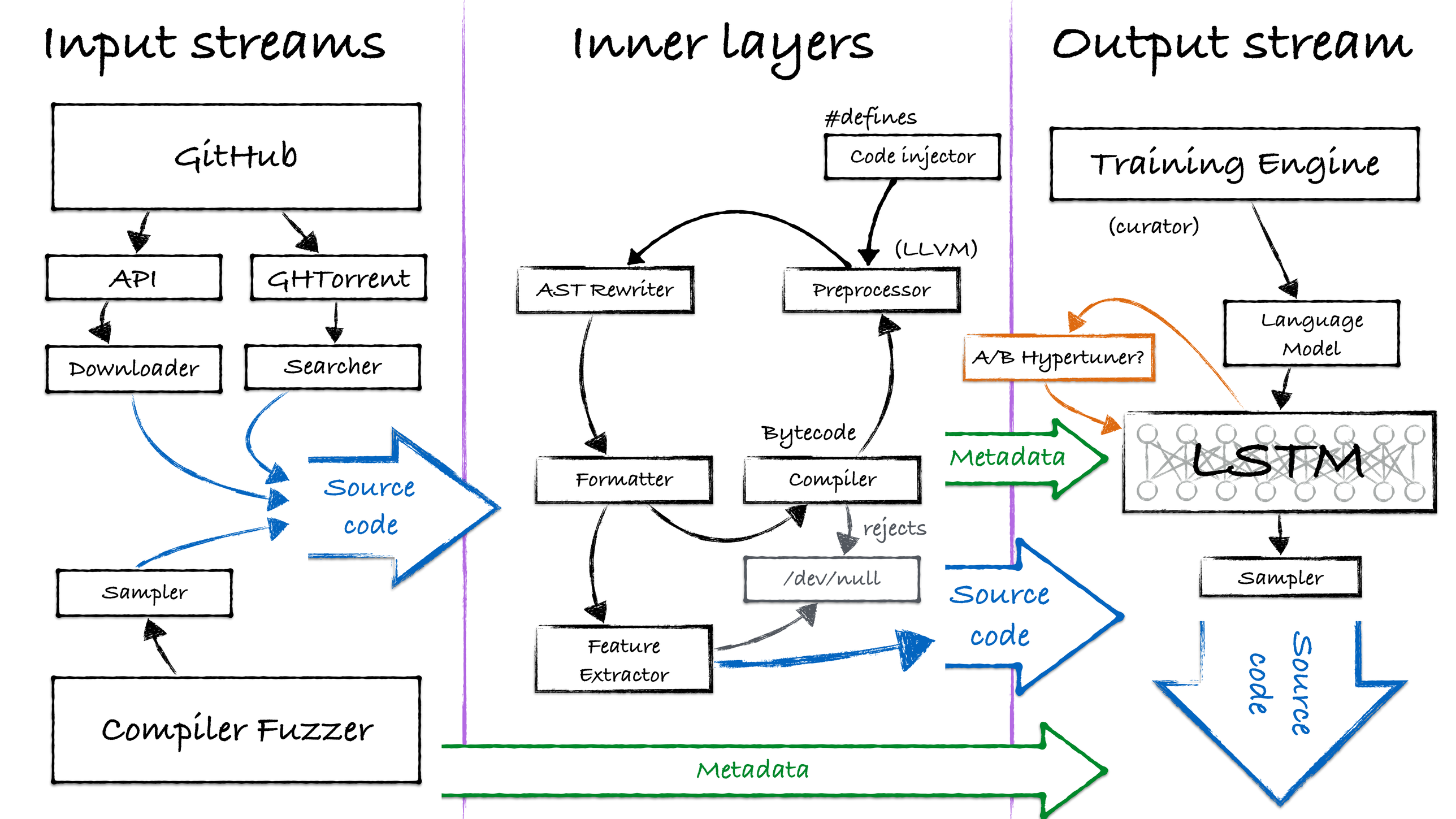
Thanks for listening!

Representative benchmarks are hugely important in compiler development.

GitHub enables access to massive datasets of real world code.

Deep Learning can successfully learn complex structures even at the character level.

Chris Cummins **http://chriscummins.cc**


Input streams
Inner layers
Output stream
GitHub
API
GHTorrent
Downloader
Searcher
Source code
Sampler
Compiler Fuzzer
#defines
Code injector
(LLVM)
AST Rewriter
Preprocessor
Bytecode
Formatter
Compiler
rejects
/dev/null
Feature Extractor
Metadata
Source code
Training Engine
(curator)
Language Model
A/B Hypertuner?
LSTM
Metadata
Sampler
Source code