CS 598sm

Probabilistic & Approximate Computing

http://misailo.web.engr.Illinois.edu/courses/cs598

(Recap)
Approximate Program Safety:
Information-flow Type Systems
Relational Logic Reasoning

EnerJ Type System

Idea:

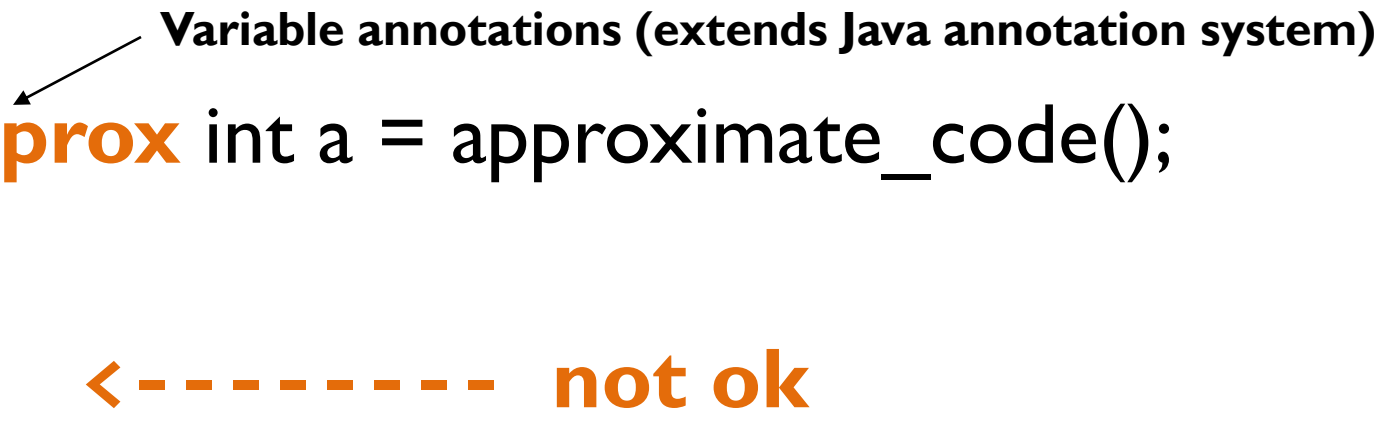
Isolate code and data that **must be precise** from those that **can be approximated**

Sampson, Dietl, Fortuna, Gnanapragasam, Ceze, Grossman
EnerJ: Approximate Data Types for Safe and General Low-Power Computation (PLDI 2011)

EnerJ Type System

Idea:

Isolate code and data that **must be precise** from those that **can be approximated**

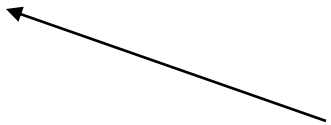


EnerJ Type System

Idea:

Isolate code and data that **must be precise** from those that **can be approximated**

```
@Approx int a = approximate_code();
int p;
if (a > 3) { p = 1; } else { p = 2; }
```



Control flow dependency (implicit flow)

EnerJ Type System

Idea:

Isolate code and data that **must be precise** from those that **can be approximated**

```
@Approx int a = approximate_code();
int p;
p = endorse(a);    <-------- ok
```

Like "(cast_type) a" in Java

EnerJ Type System

Consequence:

Then the approximate parts may be optimized automatically, but the developer needs to **ensure the endorsed values are valid**.

```
@Approx int a = approximate_code();
int p;
p = endorse(a);    <-------- ok
if ( isValid(p) ) { … } else { errorHandle(a) }
```

EnerJ Type System

Motivation:

Security information flow type systems –
prevent the program from leaking information about **private** variables into **public** variables.

Noninterference [Goguen and Meseguer 1982]:

"one group of users, using a certain set of commands is **noninterfering** with another group of users if the first group does with those commands can no effect on what the second group of users can see."

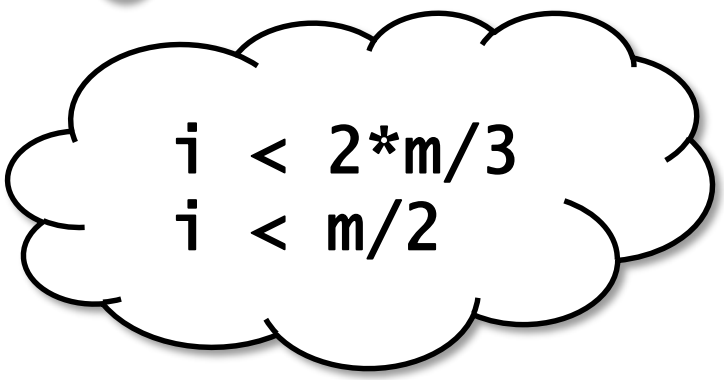
General Formal Reasoning About Relaxed Programs

Carbin, Kim, Misailovic, Rinard
Proving acceptability properties of relaxed nondeterministic approximate programs (PLDI'12)

Carbin, Kim, Misailovic, Rinard
Verified integrity properties for safe approximate program transformations (PEPM'13)

Relational Safety Verification

```
for (i=0; i < m; i++) {
  sum = sum + x[i]
}

avg = sum / m
```



Carbin, Kim, Misailovic, Rinard
Proving acceptability properties of relaxed nondeterministic approximate programs (PLDI'12)
Carbin, Kim, Misailovic, Rinard
Verified integrity properties for safe approximate program transformations (PEPM'13)

Relational Safety Verification

```
relax (m) st (0 < m <= old(m))

for (i=0; i < m; i++) {
  sum = sum + x[i]
}

avg = sum / m
```

Relational Safety Verification

```
relax (m) st (0 < m <= old(m))

for (i=0; i < m; i++) {
  sum = sum + x[i]
}

avg = sum / m
```

Transformed execution accesses only (a subset of) memory locations that the original execution would have accessed

Relational Safety Verification

```
relax (m) st (0 < m <= old(m))

for (i=0; i < m; i++) {
  sum = sum + x[i]
}

avg = sum / m
```

The difference between the variable in the original and approximate runs is at most δ

$$|sum\langle o\rangle - sum\langle r\rangle| \leq \delta$$

Relative Safety

If the original program **satisfies all assertions**,
then the relaxed program satisfies all assertions

Relative Safety vs. Just Safety

Established **through any means**:
verification, testing, code review

If the original program **satisfies all assertions**,
then the relaxed program satisfies all assertions

Any inconsistent behavior must be in the original program!

Relative Safety vs. Just Safety

Established **through any means**:
verification, testing, code review

If the original program **satisfies all assertions**,
then the relaxed program satisfies all assertions

General Proofs: Mechanized in Coq [PLDI '12]

Pointer Safety: Automatic for loop perforation [PEPM '13]

NUMBER REPRESENTATION

Numb3rs

Integers vs Machine Integers

- Precision
- Signed/unsigned

Reals vs Rationals vs Floats

- Precision
- Special values

Complex numbers etc.

Numb3rs

Dynamic range: the range of representable numbers.

Important to consider: number of values that can be represented within the dynamic range

Precision / resolution: the distance between two represented numbers

Numb3rs

Problems with finite representations:

- Overflows
- Underflows
- Infitinies
- NaN (not a number)
- …

Rounding Error

Difference between results obtained between the exact solution (using the mathematical representation) and the finite-space representation of numbers

Machine epsilon: measure of roundoff error level

$\pi \rightarrow 3.14$

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

On a side…

https://docs.python.org/3/tutorial/floatingpoint.html

```
>>> from decimal import Decimal
>>> from fractions import Fraction

>>> Fraction.from_float(0.1) Fraction(3602879701896397, 36028797018963968)

>>> (0.1).as_integer_ratio() (3602879701896397, 36028797018963968)

>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')

>>> (Decimal.from_float(0.1), '.17') '0.10000000000000001'
```

NUMERICAL APPROXIMATIONS

Error Metric

For idealized computation P (running on idealized input x) and approximate computation P' (running on the approximated input x':

$$Err = \max_{x,x'} | P(x) - P'(x') |$$

Algorithmic Approximation

How to compute sin(x) ?

Taylor Series (1715)

$$f(a) + \frac{f'(a)}{1!}(x\text{-}a)$$

$$+ \frac{f''(a)}{2!}(x\text{-}a)^2 + \frac{f'''(a)}{3!}(x\text{-}a)^3$$

$$+ \cdots,$$

Algorithmic Approximation

$$\boldsymbol{sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}}$$

What is the approximation error?

Algorithmic Approximation

$$\boldsymbol{sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}}$$

$$\boldsymbol{err < \frac{|x^9|}{9!}}$$

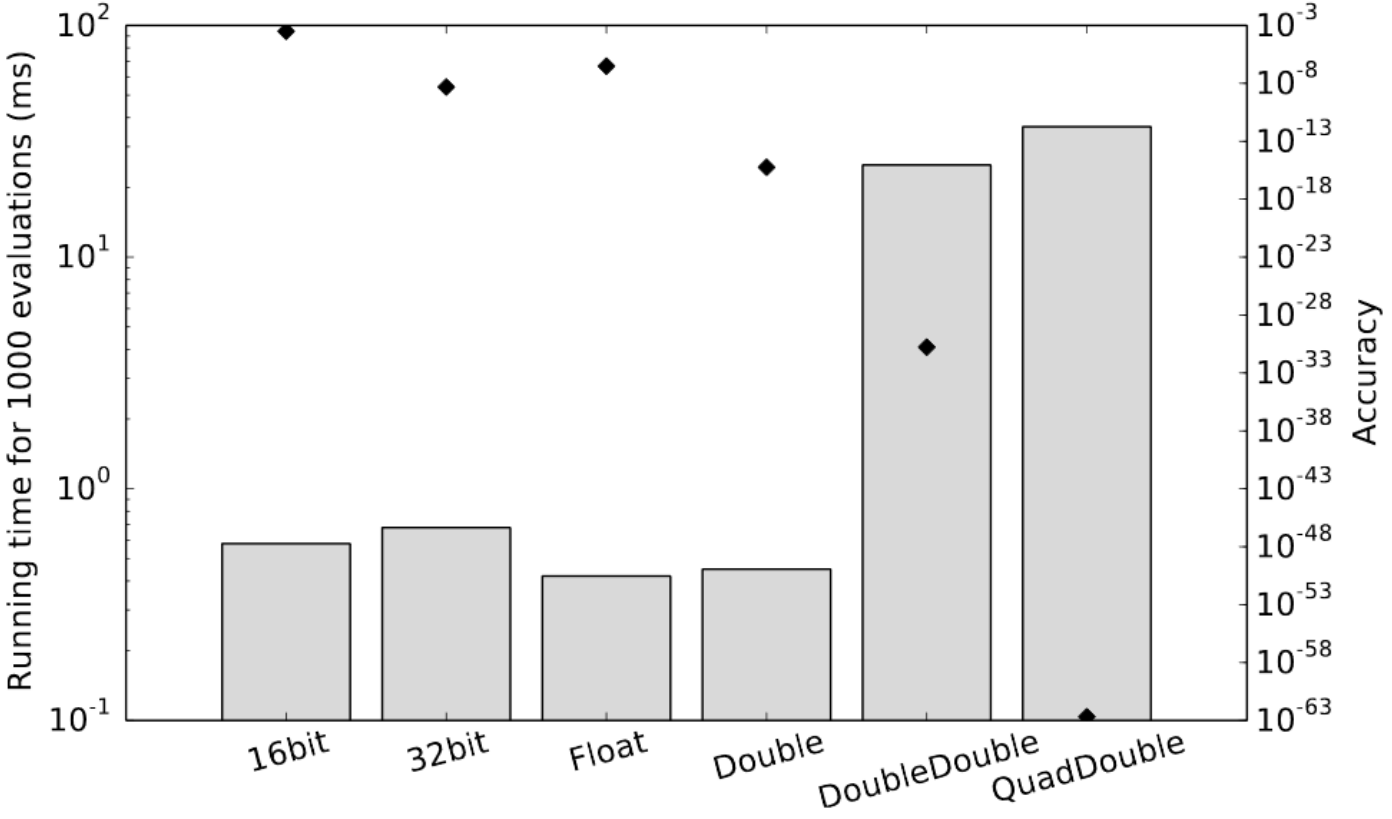
Algorithmic Approximation

$$\boldsymbol{sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}}$$

$$\boldsymbol{err < \frac{|x^9|}{9!}}$$

Where's the catch?

```
def sineWithError(x: Real): Real = {
  require(x > -1.57079632679 && x < 1.57079632679 && x +/- 1e-11)

  x - (x*x*x)/6.0 + (x*x*x*x*x)/120.0 - (x*x*x*x*x*x*x)/5040.0
} ensuring(res => res +/- 1.001e-11)
```



Towards a Compiler for Reals (TOPLAS 2017)

Other options

Orthogonal-basis polynomials: e.g., Chebyshev polynomials can approximate the function to the desired precision on the entire interval

Rational functions

Splines: piecewise function, each piece is a polynomial

Try out: https://www.chebfun.org/

Real Implementation

```
/*****************************************************************/
/* An ultimate sin routine. Given an IEEE double machine number x
*/
/* it computes the correctly rounded (to nearest) value of sin(x)  */
/*****************************************************************/
#ifndef IN_SINCOS
double SECTION __sin (double x)
{
  double t, a, da;   mynumber u;   int4 k, m, n;   double retval = 0;
  SET_RESTORE_ROUND_53BIT (FE_TONEAREST);

  u.x = x;
  m = u.i[HIGH_HALF];
  k = 0x7fffffff & m;            /* no sign           */
  if (k < 0x3e500000) {          /* if x->0 =>sin(x)=x */
      math_check_force_underflow (x);
      retval = x;
    }
/*--------------------------- 2^-26<|x|< 0.855469-------------------
-- */
  else if (k < 0x3feb6000) {
      /* Max ULP is 0.548.  */
      retval = do_sin (x, 0);
    }                                                          /*
else  if (k < 0x3feb6000)    */

/*----------------------- 0.855469  <|x|<2.426265  ----------------
----*/
  else if (k < 0x400368fd)
    {
      t = hp0 - fabs (x);
      /* Max ULP is 0.51.  */
      retval = copysign (do_cos (t, hp1), x);
    }                                                          /*
else  if (k < 0x400368fd)    */

/*-------------------------- 2.426265<|x|< 105414350 ----------------
------*/
  else if (k < 0x419921FB)
    {
      n = reduce_sincos (x, &a, &da);
      retval = do_sincos (a, da, n);
    }                                                          /*
else  if (k <  0x419921FB )    */

/* --------------------105414350 <|x| <2^1024-----------------------
------*/
  else if (k < 0x7ff00000)
    {
      n = __branred (x, &a, &da);
      retval = do_sincos (a, da, n);
    }
/*--------------------- |x| > 2^1024 ----------------------------------
--*/
  else
    {
      if (k == 0x7ff00000 && u.i[LOW_HALF] == 0)
                __set_errno (EDOM);
      retval = x / x;
    }

  return retval;
}
```

https://sourceware.org/git/?p=glibc.git;a=blob;f=sysdeps/ieee754/dbl-64/s_sin.c;hb=HEAD#l281

Real Implementation (More!)

```
/* Given a number partitioned into X and DX, this function computes the sine of
   the number by combining the sin and cos of X (as computed by a variation of
   the Taylor series) with the values looked up from the sin/cos table to get
   the result.  */
static __always_inline double do_sin (double x, double dx)  {
  double xold = x;
  /* Max ULP is 0.501 if |x| < 0.126, otherwise ULP is 0.518.  */
  if (fabs (x) < 0.126)  return TAYLOR_SIN (x * x, x, dx);

  mynumber u;
  if (x <= 0) dx = -dx;
  u.x = big + fabs (x);
  x = fabs (x) - (u.x - big);

  double xx, s, sn, ssn, c, cs, ccs, cor;
  xx = x * x;
  s = x + (dx + x * xx * (sn3 + xx * sn5));
  c = x * dx + xx * (cs2 + xx * (cs4 + xx * cs6));
  SINCOS_TABLE_LOOKUP (u, sn, ssn, cs, ccs);
  cor = (ssn + s * ccs - sn * c) + cs * s;
  return copysign (sn + cor, xold);
}
```

…and this is not all!

Sensitivity

So far we talked about the error inside the computation

How does that error propagate?

Sensitivity

If input x changes by δ
by how much the output of f(x) changes?

$$F_1(x) = x + 1$$

$$F_2(x) = x^2 + 1$$

$$F_3(x) = e^x$$

Lipschitz Continuity

Sets a linear bound on error propagation:

$$\forall x_1, x_2 \ . \ |f(x_1) - f(x_2)| \leq K \cdot |x_1 - x_2|$$

Locally Lipschitz continuous in neighborhood U

$$\forall x_1, \forall x_2 \in U(x_1) \ . \ |f(x_1) - f(x_2)| \leq K \cdot |x_1 - x_2|$$

Tuning Floating Point Programs: Precimonious

Key idea:

- Identify operations for which, when approximated the output is sensitive to change
- Do not reduce their precision, try other instructions

Delta debugging: make multiple changes, then reduce and split the sets if some of the variables cause low accuracy

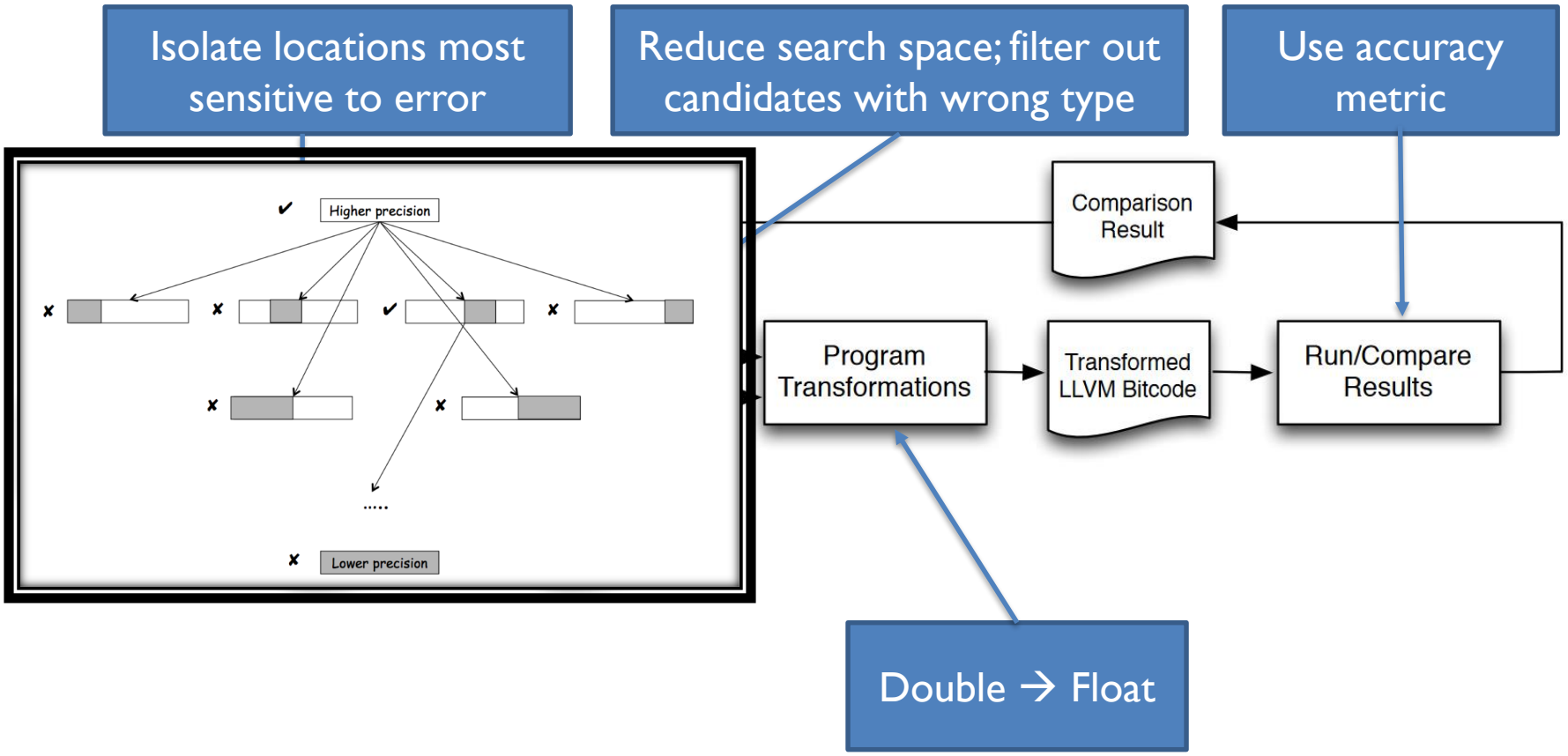
Precimonius: Tuning Assistant for Floating-Point Precision; Rubio-Gonzalez et al. SC 2013

Precimonious Example

```
1  long double fun( long double x ) {
2    int k, n = 5;
3    long double t1;
4    long double d1 = 1.0L;
5
6    t1 = x;
7    for( k = 1; k <= n; k++ ) {
8      d1 = 2.0 * d1;
9      t1 = t1 + sin (d1 * x) / d1;
10   }
11   return t1;
12 }
13
14 int main( int argc, char **argv) {
15   int i, n = 1000000;
16   long double h, t1, t2, dppi;
17   long double s1;
18
19   t1 = -1.0;
20   dppi = acos(t1);
21   s1 = 0.0;
22   t1 = 0.0;
23   h = dppi / n;
24
25   for( i = 1; i <= n; i++ ) {
26     t2 = fun (i * h);
27     s1 = s1 + sqrt (h*h + (t2 - t1)*(t2 - t1));
28     t1 = t2;
29   }
30   // final answer is stored in variable s1
31   return 0;
32 }
```

```
double fun( double x ) {
  int k, n = 5;
  double t1;
  float d1 = 1.0f;

  t1 = x;
  for( k = 1; k <= n; k++ ) {
    d1 = 2.0 * d1;
    t1 = t1 + sin (d1 * x) / d1;
  }
  return t1;
}

int main( int argc, char **argv) {
  int i, n = 1000000;
  double h, t1, t2, dppi;
  long double s1;

  t1 = -1.0;
  dppi = acos(t1);
  s1 = 0.0;
  t1 = 0.0;
  h = dppi / n;

  for( i = 1; i <= n; i++ ) {
    t2 = fun (i * h);
    s1 = s1 + sqrt (h*h + (t2 - t1)*(t2 - t1));
    t1 = t2;
  }
  // final answer is stored in variable s1
  return 0;
```

Precimonious

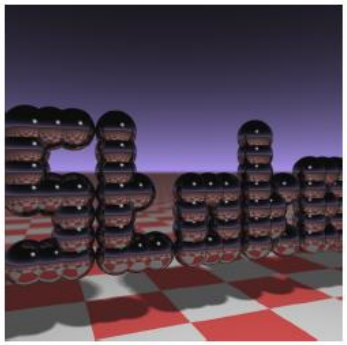


Precimonius: Tuning Assistant for Floating-Point Precision; Rubio-Gonzalez et al. SC 2013

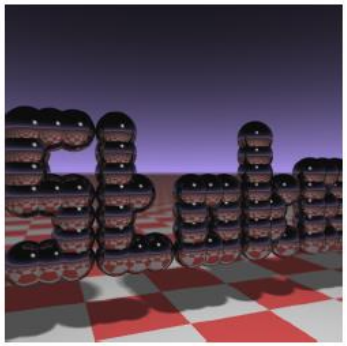
Stoke

- Superoptimizer: tries various ordering of instructions
- Stochastic: searches for the regions of programs and instructions that may have better chance of giving high performance using MCMC

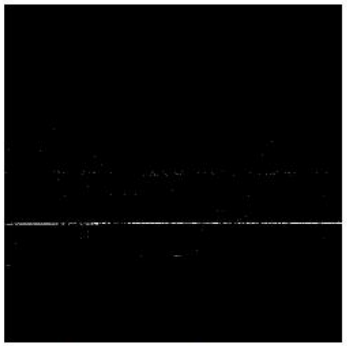
http://stoke.stanford.edu



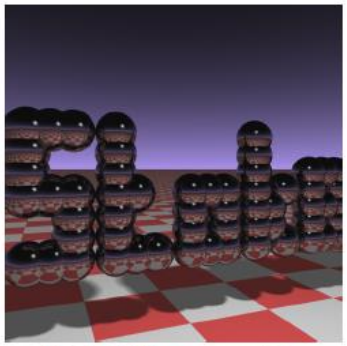
(a) Bit-wise correct (30.2%)



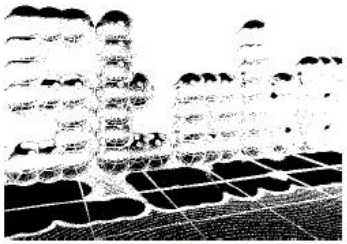
(b) Valid lower precision (36.6%)



(c) Error pixels (shown white)



(d) Invalid lower precision



(e) Error pixels (shown white)

Stochastic Optimization of Floating-Point Programs with Tunable Precision (Schkufza et al. PLDI 2014)

Machine Learning: Quantization in inference

FP32 → FP16

- Float to half-float

FP32 → INT8

- From 10^38 values to 256

FP32 → Bool

- Extreme, but works for problems like MNIST, CIFAR-10
  Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1 (2016) Bengio et al.

Machine Learning: Cost of Operations

INT8 Operation	Energy Saving vs FP32	Area Saving vs FP32
Add	30x	116x
Multiply	18.5x	27x

William Dally. High-Performance Hardware for Machine Learning. Tutorial, NIPS, 2015

But also consider the cost of transferring data

Other Aggressive Optimizations

- Training and Re-Training with quantization in mind (add losses expected by quantization)
- Replacing the activation function: RELU is unbounded, use a bounded one instead
- Network architecture changes: to accommodate quantized weights
- Iterative quantization: quantize only a part of the network
- Neural network pruning: compress the network and remove both nodes and edges
- Neural network perforation: compute outputs based on only some of the inputs of the weights of the network

More details at https://nervanasystems.github.io/distiller/quantization.html

Considerations for Training

Training high accuracy – inference low accuracy

- Stochastic gradient descent more sensitive to quantization
- Postprocess highly accurate (FP32) network

But some interesting new devlopments:

- Training Deep Neural Networks with 8-bit Floating Point Numbers (NeurIPS 2018)
- The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks, Frankle and Carbin (ICLR 2019)