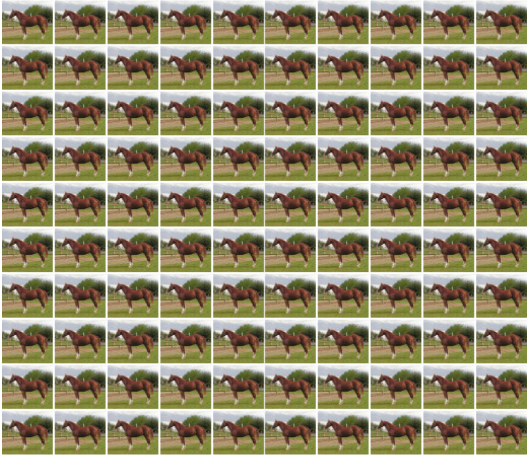
Lecture 23 — Password Cracking, Reduced-Resource Computation

Patrick Lam
patrick.lam@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 15, 2018

Saw principles of OpenCL programming:



(Credit: Karlyne, Wikimedia Commons)

Key concepts: kernels, buffers.

Part I

GPU Application: Password Cracking

Reference: scrypt

scrypt is the algorithm behind DogeCoin.

The reference:

Colin Percival, “Stronger Key Derivation via Sequential Memory-Hard Functions”.
Presented at BSDCan’09, May 2009.

`http://www.tarsnap.com/scrypt.html`

- **not** plaintext!
- hashed and salted

One-way function:

- $x \mapsto f(x)$ easy to compute; but
- $f(x) \overset{?}{\mapsto} x$ hard to reverse.

Examples: SHA1, Scrypt.

Breaking the Hash

How can we reverse the hash function?

- Brute force.

GPUs (or custom hardware) are good at that!

The Arms Race: Making Cracking Difficult

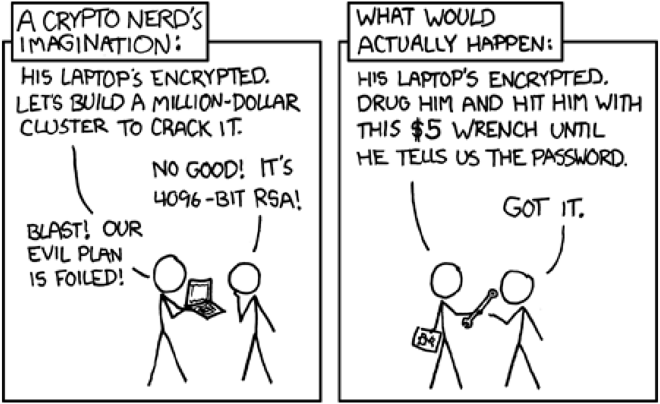
Historically: force repeated iterations of hashing.

Main idea behind scrypt (hence DogeCoin):

- make hashing expensive in time and space.

Implication: require more circuitry to break passwords.
Increases both # of operations and cost of brute-forcing.

Of course, there's always this form of cracking:



(Source: xkcd 538)

Formalizing “expensive in time and space”

Definition

A memory-hard algorithm on a Random Access Machine is an algorithm which uses $S(n)$ space and $T(n)$ operations, where $S(n) \in \Omega(T(n)^{1-\varepsilon})$.

Such algorithms are expensive to implement in either hardware or software.

Generalizing to functions

Next, add a quantifier:
move from particular algorithms to underlying functions.

A *sequential memory-hard function* is one where:

- the fastest sequential algorithm is memory-hard; and
- it is impossible for a parallel algorithm to asymptotically achieve lower cost.

Exhibit. ReMix is a concrete example of a sequential memory hard function.

The paper concludes with an example of a more realistic (cache-aware) model and a function in that context, BlockMix.

Part II

Reduced-Resource Computation

Trading Accuracy for Performance

Consider Monte Carlo integration.
It illustrates a general tradeoff: accuracy vs performance.
You'll also implement this tradeoff manually in A4
(with domain knowledge).

Martin Rinard generalized the accuracy vs performance tradeoff with:

- early phase termination [OOPSLA07]
- loop perforation [CSAIL TR 2009]

Early Phase Termination

We've seen barriers before.
No thread may proceed past a barrier until all of the threads reach the barrier.

This may slow down the program: maybe one of the threads is horribly slow.

Solution: kill the slowest thread.

"Oh no, that's going to change the meaning of the program!"

Early Phase Termination: When is it OK anyway?

OK, so we don't want to be completely crazy.

Instead:

- develop a statistical model of the program behaviour.
- only kill tasks that don't introduce unacceptable distortions.

When we run the program:

get the output, plus a confidence interval.

Early Phase Termination: Two Examples

Monte Carlo simulators:
Raytracers:

- already picking points randomly.

In both cases: spawn a lot of threads.

Could wait for all threads to complete;
or just compensate for missing data points,
assuming they look like points you did compute.

Early Phase Termination: Another Justification

In scientific computations:

- using points that were measured (subject to error);
- computing using machine numbers (also with error).

Computers are only providing simulations, not ground truth.

Actual question: is the simulation is good enough?

Loop Perforation

Like early-phase termination, but for sequential programs:
throw away data that's not actually useful.

```
for (i = 0; i < n; ++i) sum += numbers[i];
```

$\Downarrow$

```
for (i = 0; i < n; i += 2) sum += numbers[i];
sum *= 2;
```

This gives a speedup of ~ 2 if `numbers[]` is nice.

Works for video encoding: can't observe difference.

Applications of Reduced Resource Computation

Loop perforation works for:

- evaluating forces on water molecules (summing numbers);
- Monte-Carlo simulation of swaption pricing;
- video encoding.

More on the video encoding example:
Changing loop increments from 4 to 8 gives:

- speedup of 1.67;
- signal-to-noise ratio decrease of 0.87%;
- bitrate increase of 18.47%;
- visually indistinguishable results.

Video Encoding Skeleton Code

```
min = DBL_MAX;
index = 0;
for (i = 0; i < m; i++) {
  sum = 0;
  for (j = 0; j < n; j++) sum += numbers[i][j];
  if (min < sum) {
    min = sum;
    index = i;
  }
}
```

The optimization changes the loop increments and then compensates.