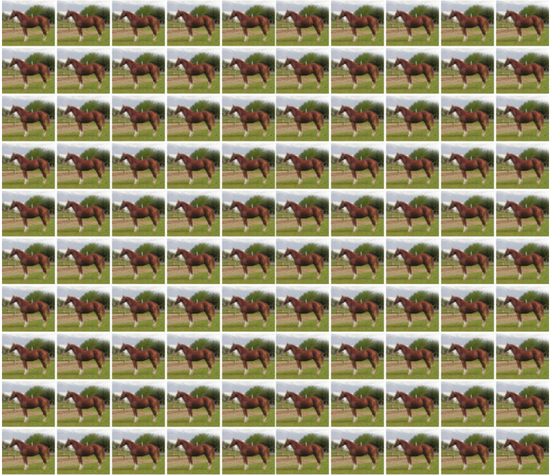
Lecture 34—Password cracking on GPUs; reduced-resource computation; STM

ECE 459: Programming for Performance

March 30, 2015

Last Time: OpenCL

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`

Acceptable Practices for Password Storage

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

Cryptographic hashing

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.

Formalizing “expensive in time and space”

Want to force use of the “most memory possible” for a given number of operations.

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.

Not made of unicorn tears

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.

Early Phase Termination: Objection

"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];
```

⇓

```
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.

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.

Part III

Software Transactional Memory

STM: Introduction

Instead of programming with locks, we have transactions on memory.

- Analogous to database transactions

An old idea; recently saw some renewed interest.

A series of memory operations either all succeed; or
all fail (and get rolled back), and are later retried.

STM: Benefits

Simple programming model: need not worry about lock granularity or deadlocks.

Just group lines of code that should logically be one operation in an `atomic` block!

It is the responsibility of the implementer to ensure the code operates as an atomic transaction.

STM: Implementing a Motivating Example

```
transfer_funds(Account* sender, Account* receiver,
               double amount) {
  atomic {
    sender->funds -= amount;
    receiver->funds += amount;
  }
}
```

[Note: bank transfers aren't actually atomic!]

With locks we have two main options:

- Lock everything to do with modifying accounts
  (slow; may forget to use lock).
- Have a lock for every account
  (deadlocks; may forget to use lock).

With STM, we do not have to worry about remembering to acquire locks, or about deadlocks.

STM: Drawbacks

Rollback is key to STM.
    But, some things cannot be rolled back.
    (write to the screen, send packet over network)

Nested transactions.
    What if an inner transaction succeeds,
yet the transaction aborts?

Limited transaction size:
    Most implementations (especially all-hardware)
have a limited transaction size.

Basic STM Implementation (Software)

In all atomic blocks, record all reads/writes to a log.

At the end of the block, running thread verifies that no other threads have modified any values read.

If validation is successful, changes are **committed**. Otherwise, the block is **aborted** and re-executed.

Note: Hardware implementations exist too.

Basic STM Implementation Issues

Since you don't protect against dataraces (just rollback), a datarace may trigger a fatal error in your program.

```
atomic {
  x++;
  y++;
}
```

```
atomic {
  if (x != y)
      while (true) { }
}
```

In this silly example, assume initially `x = y`. You may think the code will not go into an infinite loop, but it can.

STM Implementations

Note: Typically STM performance is no worse than twice as slow as fine-grained locks.

- Toward.Boost.STM (C++)
- SXM (Microsoft, C#)
- Built-in to the language (Clojure, Haskell)
- AtomJava (Java)
- Durus (Python)

STM Summary

Software Transactional Memory provides a more natural approach to parallel programming:

no need to deal with locks and their associated problems.

Currently slow,

but a lot of research is going into improving it. (futile?)

Operates by either completing an atomic block,
or retrying (by rolling back) until it successfully completes.

Part IV

Overall Summary

Today's Lecture

Three topics:

- Password cracking with GPUs.
- Reduced-Resource Computation.
- High-Performance Languages.