Speeding up characteristic 2:

I. Linear maps

II. The $M(n)$ game

III. Batching

IV. Normal bases

D. J. Bernstein
University of Illinois at Chicago

NSF ITR–0716498

Part I. Linear maps

Consider computing

$h_0 = q_0$;

$h_1 = q_1$;

$h_2 = q_2 \oplus (p_0 \oplus q_0 \oplus r_0)$;

$h_3 = (p_1 \oplus q_1 \oplus r_1)$;

$h_4 = (p_2 \oplus q_2 \oplus r_2) \oplus r_0$;

$h_5 = r_1$;

$h_6 = r_2$.

Easy: 8 additions.
Can find these 8 additions
in several papers.
But 8 is not optimal!

"Wasting brain power
is bad for the environment."
Use existing algorithms
to find addition chains.

Apply, e.g., greedy additive
CSE algorithm from 1997 Paar:

- find input pair i_0, i_1
  with most popular $i_0 \oplus i_1$;
- compute $i_0 \oplus i_1$;
- simplify using $i_0 \oplus i_1$;
- repeat.

This algorithm finds repeated
$q_2 \oplus r_0$; uses 7 additions.

A new algorithm: “xor largest.”
Start with the matrix mod 2
for the desired linear map.

If two largest rows
have same first bit,
replace largest row
by its xor with
second-largest row.

Otherwise change largest row
by clearing first bit.

In both cases,
compute result recursively,
and finish with one xor.

A small example:

$$1011 = x_0 + x_2 + x_3$$
$$1111 = x_0 + x_1 + x_2 + x_3$$
$$0110 = x_1 + x_2$$
$$0101 = x_1 + x_3$$

Replace largest row
by its xor with
second-largest row.

Recursively compute

$$1011 = x_0 + x_2 + x_3$$
$$0100 = x_1 \leftarrow$$
$$0110 = x_1 + x_2$$
$$0101 = x_1 + x_3$$

plus 1 xor

of first output

into second output.

Recursively compute

0011 $\leftarrow$

0100

0110

0101

plus 1 input load, 2 xors.

Recursively compute

0011

0100

0011 $\leftarrow$

0101

plus 1 input load, 3 xors.

Recursively compute

0011

0100

0011

0001 $\leftarrow$

plus 1 input load, 4 xors.

Recursively compute

0011

0000 $\leftarrow$

0011

0001

plus 2 input loads, 4 xors.

Note: this was just a copy.

Recursively compute

$0000 \leftarrow$

0000

0011

0001

plus 2 input loads, 4 xors.

Recursively compute

0000

0000

0001 $\leftarrow$

0001

plus 3 input loads, 5 xors.

Recursively compute

0000

0000

0000 $\leftarrow$

0001

plus 3 input loads, 5 xors.

Recursively compute

0000

0000

0000

0000 $\leftarrow$

plus 4 input loads, 5 xors.

Memory friendliness:
Algorithm writes only
to the output registers.
No temporary storage.

n inputs, n outputs:
total $2n$ registers
with 0 loads, 0 stores.

Or $n+1$ registers
with n loads, 0 stores:
each input is read only once.

Or n registers
with n loads, 0 stores,
if platform has load-xor insn.

Two-operand friendliness:
Platform with $a \leftarrow a \oplus b$
but without $a \leftarrow b \oplus c$
uses only n extra copies.

Naive column sweep also uses
$n + 1$ registers, n loads,
but usually many more xors.

Input partitioning
(e.g., 1956 Lupanov) uses
somewhat more xors, copies;
somewhat more registers.

Greedy additive CSE uses
somewhat fewer xors but
many more copies, registers.

For m inputs and n outputs, average $n \times m$ matrix:

The xor-largest algorithm uses $\approx mn/\lg n$ two-operand xors; n copies; m loads; $n+1$ regs.

For m inputs and n outputs, average $n \times m$ matrix:

The xor-largest algorithm uses $\approx mn/\lg n$ two-operand xors; n copies; m loads; $n+1$ regs.

Pippenger's algorithm uses $\approx mn/\lg mn$ three-operand xors but seems to need many regs.

Pippenger proved that his algebraic complexity was near optimal for most matrices (at least without mod 2), but didn't consider regs, two-operand complexity, etc.

Our original example:

000100000

000010000

100101100

010010010

001001101

000000010

000000001

Each row has coefficients of

$p_0, p_1, p_2, q_0, q_1, q_2, r_0, r_1, r_2$.

Our original example:

000100000

000010000

000101100 $\leftarrow$

010010010

001001101

000000010

000000001

plus 1 xor, 1 input load.

Our original example:

000100000

000010000

000101100

000010010 $\leftarrow$

001001101

000000010

000000001

plus 2 xors, 2 input loads.

Our original example:

000100000

000010000

000101100

000010010

000001101 $\leftarrow$

000000010

000000001

plus 3 xors, 3 input loads.

Our original example:

000100000

000010000

000001100 $\leftarrow$

000010010

000001101

000000010

000000001

plus 4 xors, 3 input loads.

Our original example:

000000000 $\leftarrow$

000010000

000001100

000010010

000001101

000000010

000000001

plus 4 xors, 4 input loads.

Our original example:

000000000

000010000

000001100

000000010 $\leftarrow$

000001101

000000010

000000001

plus 5 xors, 4 input loads.

Our original example:

000000000

000000000 $\leftarrow$

000001100

000000010

000001101

000000010

000000001

plus 5 xors, 5 input loads.

Our original example:

000000000

000000000

000001100

000000010

000000001 $\leftarrow$

000000010

000000001

plus 6 xors, 5 input loads.

Our original example:

000000000

000000000

000000100 $\leftarrow$

000000010

000000001

000000010

000000001

plus 7 xors, 6 input loads.

Our original example:

000000000

000000000

000000000 $\leftarrow$

000000010

000000001

000000010

000000001

plus 7 xors, 7 input loads.

Our original example:

000000000

000000000

000000000

000000000 $\leftarrow$

000000001

000000010

000000001

plus 7 xors, 7 input loads.

Our original example:

000000000

000000000

000000000

000000000

000000001

000000000 $\leftarrow$

000000001

plus 7 xors, 8 input loads.

Our original example:

000000000

000000000

000000000

000000000

000000000 $\leftarrow$

000000000

000000001

plus 7 xors, 8 input loads.

Our original example:

000000000

000000000

000000000

000000000

000000000

000000000

000000000 $\leftarrow$

plus 7 xors, 9 input loads.

Algorithm found the speedup.

Part II. The $M(n)$ game

Define $M(n)$
as the minimum number of
bit operations (ands, xors)
needed to multiply
n-bit polys $f, g \in \mathbf{F}_2[x]$
(in standard representation).

e.g. $M(2) \le 5$:
to compute
$h_0 + h_1 x + h_2 x^2 =$
$(f_0 + f_1 x)(g_0 + g_1 x)$
can compute $h_0 = f_0 g_0$,
$h_1 = f_0 g_1 + f_1 g_0$, $h_2 = f_1 g_1$
with 4 ands, 1 xor.

Schoolbook multiplication:
$M(n) \le \Theta(n^2)$.

1963 Karatsuba:
$M(n) \le \Theta(n^{\lg 3})$.

1963 Toom:
$M(n) \le n2^{\Theta(\sqrt{\lg n})}$.

1971 Schönhage–Strassen:
$M(n) \le \Theta(n \lg n \lg \lg n)$.

2007 Fürer
improves $\lg \lg n$ for integers
but doesn't help mod 2.

What does this tell us
about $M(131)$ or $M(251)$?

Absolutely nothing!

Reanalyze algorithms
to see exact complexity.

Rethink algorithm design
to find constant-factor
(and sub-constant-factor)
speedups that are not
visible in the asymptotics.

Schoolbook recursion:

$M(n+1) \le M(n) + 4n$.

Hence $M(n) \le 2n^2 - 2n + 1$.

Karatsuba recursion
as commonly stated:

$M(2n) \le 3M(n) + 8n - 4$.

e.g. Karatsuba for $n = 1$:

$f = f_0 + f_1 x$,

$g = g_0 + g_1 x$,

$h_0 = f_0 g_0$,

$h_2 = f_1 g_1$,

$h_1 = (f_0 + f_1)(g_0 + g_1) - h_0 - h_2$

$\Rightarrow fg = h_0 + h_1 x + h_2 x^2$.

Karatsuba for $n = 2$:

$$f = f_0 + f_1 x + f_2 x^2 + f_3 x^3,$$
$$g = g_0 + g_1 x + g_2 x^2 + g_3 x^3,$$

$$H_0 = (f_0 + f_1 x)(g_0 + g_1 x),$$
$$H_2 = (f_2 + f_3 x)(g_2 + g_3 x),$$
$$H_1 = (f_0 + f_2 + (f_1 + f_3)x) \cdot$$
$$(g_0 + g_2 + (g_1 + g_3)x)$$
$$- H_0 - H_2$$

$$\Rightarrow fg = H_0 + H_1 x^2 + H_2 x^4.$$

Initial linear computation:

$f_0 + f_2, f_1 + f_3, g_0 + g_2, g_1 + g_3$;

cost 4.

Three size-2 mults producing

$H_0 = q_0 + q_1 x + q_2 x^2$;

$H_2 = r_0 + r_1 x + r_2 x^2$;

$H_0 + H_1 + H_2 = p_0 + p_1 x + p_2 x^2$.

Final linear reconstruction:

$$H_1 = (p_0 - q_0 - r_0) + (p_1 - q_1 - r_1)x + (p_2 - q_2 - r_2)x^2,$$

cost 6;

$fg = H_0 + H_1 x^2 + H_2 x^4$,

cost 2.

Let's look more closely

at the reconstruction:

$fg = h_0 + h_1 x + \cdots + h_6 x^6$ with

$$h_0 = q_0;$$
$$h_1 = q_1;$$
$$h_2 = q_2 + (p_0 - q_0 - r_0);$$
$$h_3 = (p_1 - q_1 - r_1);$$
$$h_4 = (p_2 - q_2 - r_2) + r_0;$$
$$h_5 = r_1;$$
$$h_6 = r_2.$$

Let's look more closely

at the reconstruction:

$fg = h_0 + h_1 x + \cdots + h_6 x^6$ with

$h_0 = q_0;$

$h_1 = q_1;$

$h_2 = q_2 + (p_0 - q_0 - r_0);$

$h_3 = (p_1 - q_1 - r_1);$

$h_4 = (p_2 - q_2 - r_2) + r_0;$

$h_5 = r_1;$

$h_6 = r_2.$

We've seen this before!

Reduce $6 + 2 = 8$ ops to 7 ops

by reusing $q_2 - r_0$.

2000 Bernstein:
$M(2n) \le 3M(n) + 7n - 3.$

2009 Bernstein:
new bounds on $M(n)$
from further improvements
to Karatsuba, Toom, etc.
binary.cr.yp.to/m.html

Typically 20% smaller than
2003 Rodríguez-Henríquez–Koç,
2005 Chang–Kim–Park–Lim,
2006 Weimerskirch–Paar,
2006 von zur Gathen–Shokrollahi,
2007 Peter–Langendörfer.

So far have focused on
$M(n)$ for small n,
but different techniques
are better for large n.

I'm now exploring impact
of 2008 Gao–Mateer.

For $\mathbf{F}_2 \subseteq \mathbf{F}_q \subseteq k$:
1988 Wang–Zhu, 1989 Cantor
diagonalize $k[t]/(t^q + t)$ using
$\approx 0.5q \lg q$ mults in k,
$\approx 0.5q (\lg q)^{\lg 3}$ adds in k.
2008 Gao–Mateer use
$\approx 0.5q \lg q$ mults,
$\approx 0.25q \lg q \lg \lg q$ adds.

“Who cares?”

Conventional wisdom:

Detailed $M(n)$ analysis has very little relevance to software speed.

We multiply f by g by looking up 4 bits of f in a size-16 table of precomputed multiples of g; looking up next 4 bits; etc. One table lookup replaces many bit operations!

Might use Karatsuba etc., but only for large n.

Part III. Batching

Classic $\mathbf{F}_p^*$ index calculus needs to check smoothness of many positive integers $< p$.

Smooth integer: integer with no prime divisors $> y$.
Typical: $(\log y)^2 \in (1/2 + o(1)) \log p \log \log p$.

Many: typically $y^{2+o(1)}$, of which $y^{1+o(1)}$ are smooth.

(Modern index calculus, NFS: smaller integers; smaller y.)

How to check smoothness?

Old answers: Trial division,
time $y^{1+o(1)}$; rho, time $y^{1/2+o(1)}$, assuming standard conjectures.

Better answer: ECM etc.
Time $y^{o(1)}$; specifically
$\exp\sqrt{(2+o(1))\log y \log\log y}$,
assuming standard conjectures.

Much better answer
(in standard RAM model):
Known *batch* algorithms
test smoothness of *many*
integers simultaneously.
Time per input: $(\log y)^{O(1)}$
$= \exp O(\log\log y)$.

General pattern:

Algorithm designer optimizes algorithm for *one* input.

But algorithm is then applied to *many* inputs! Oops.

Often much better speed from *batch* algorithms optimized for many inputs.

e.g. Batch ECDL: $\sqrt{\#}$ speedup.
Batch NFS: smaller exponent.
Can find many more examples.

Surprising recent example:
Batching can save time
in *multiplication*!

Largest speedups: $\mathbf{F}_2[x]$.

Consequence: New speed record
for public-key cryptography.
37895 scalar mults/second
on a 3.2GHz Phenom II X4 for
a secure elliptic curve/$\mathbf{F}_{2^{251}}$.

http://binary.cr.yp.to

Surprising recent example:
Batching can save time
in *multiplication*!

Largest speedups: $\mathbf{F}_2[x]$.

Consequence: New speed record
for public-key cryptography.
37895 scalar mults/second
on a 3.2GHz Phenom II X4 for
a secure elliptic curve/$\mathbf{F}_{2^{251}}$.

`http://binary.cr.yp.to`

Note: No subfields were exploited
in the creation of this record.

Simplest batching technique: "bitslicing."

Transpose 128 polynomials $f_0, f_1, \ldots, f_{127} \in \mathbf{F}_2[x]$, each having d coefficients, into d vectors $F_0, F_1, \ldots, F_{d-1} \in \mathbf{F}_2^{128}$, where $F_i[j] = f_j[i]$.

Vector operation $F_1 \oplus F_{33}$ adds bit 1 of f_j to bit 33 of f_j for each i in parallel.

Bitslicing disadvantages:

Table lookups are expensive.
e.g. `tab[`f_j mod 16`]`.

Conditional branches
are expensive.

128$\times$ volume of data;
harder to avoid
load/store bottlenecks.

Transposition costs
roughly 1 cycle per byte;
frequent transposition is bad.

Bitslicing advantages:

Free bit extraction,
bit shuffling, etc.

No word-size penalty.
e.g. 128 additions of
d-bit polynomials
cost d vector xors
instead of $128 \lceil d/128 \rceil$.
Huge speedup for small d.

$\Rightarrow$ Productive synergy
with $M(n)$ techniques.

Elliptic-curve addition
$P + Q$ traditionally uses
conditional branches:
$Q = P$? $Q = -P$? etc.

2006 Bernstein: cheaply avoid
conditional branches in
$P \mapsto nP$ if $2 \neq 0$.

2007 Bernstein–Lange,
using Edwards curves:
arbitrary group ops if $2 \neq 0$.

2008 Bernstein–Lange–
Rezaeian Farashahi,
"binary Edwards curves":
arbitrary group ops if $2 = 0$.

Part IV. Normal bases

Current ECRYPT project, spearheaded by Tanja Lange: break Certicom's ECC2K-130.

i.e., compute discrete log of a challenge point on $y^2 + xy = x^3 + 1$ over $\mathbf{F}_{2^{131}}$.

Carefully selected iteration function for Pollard rho involves 5 mults, 21 squarings, 7 adds, occasional inversions, and one computation of weight in normal basis.

$\mathbf{F}_{2^{131}}$ has type-2 normal basis $\zeta + \zeta^{-1}$, $\zeta^2 + \zeta^{-2}$, $\zeta^4 + \zeta^{-4}$, $\ldots$, $\zeta^{2^{130}} + \zeta^{-2^{130}}$ where ζ is primitive 263rd root of 1.

Weight is sum of coefficients.

Squaring is rotation.

Multi-squaring is rotation.

Inversion by Fermat uses many multi-squarings.

But fast ECDL software
uses polynomial basis: e.g.,
basis $1, x, x^2, \ldots, x^{130}$ of
$\mathbf{F}_2[x]/(x^{131} + x^{13} + x^2 + x + 1)$.

Many obvious disadvantages:
more expensive squaring,
multi-squaring, inversion;
must convert to normal basis
(e.g., with xor-largest)
before computing weight.

But huge speedup in the 5 mults:
polynomial multiplication
uses Karatsuba etc.;
reduction is very fast.

How slow is normal-basis mult?

Type-1 normal basis of $\mathbf{F}_{2^n}$, where 2 has order n mod $n+1$, is a permutation of
$\zeta, \zeta^2, \ldots, \zeta^n$
in $\mathbf{F}_2[\zeta]/(\zeta^{n+1} - 1)$.

$M(n)$ operations to multiply, obtaining coefficients of
$\zeta^2, \zeta^3, \ldots, \zeta^{2n}$.

$2n - 1$ operations to reduce
$\zeta^2, \zeta^3, \ldots, \zeta^{2n}$
to $\zeta, \zeta^2, \ldots, \zeta^n$.

Alternative: $M(n+1) + n$ for redundant $1, \zeta, \ldots, \zeta^n$.

Type-2 normal basis of $\mathbf{F}_{2^n}$,
where 2 has order n mod $2n+1$,
is a permutation of
$\zeta+\zeta^{-1}$, $\zeta^2+\zeta^{-2}$,
$\zeta^3+\zeta^{-3}$, $\ldots$, $\zeta^n+\zeta^{-n}$
in $\mathbf{F}_2[\zeta]/(\zeta^{2n+1}-1)$.

2000 Gao–von zur Gathen–
Panario–Shoup:
$2M(n)+O(n)$ operations
to multiply on this basis.

Polynomial basis of $\mathbf{F}_{2^n}$
is about twice as fast.

2007 von zur Gathen–
Shokrollahi–Shokrollahi:
$M(n) + O(n \lg n)$ operations
to multiply on this basis.

2009 Bernstein:
improved variant of algorithm
sets Core 2 speed records
for the ECC2K-130 attack.

2009 Schwabe:
also Cell speed records.

2009 Bernstein–Lange:
mix normal bases
with polynomial bases
and speed up reduction.

vzG–S–S in a nutshell:

Write $N_j = \zeta^j + \zeta^{-j}$
and $P_j = (\zeta + \zeta^{-1})^j$.

If

$f_0 + f_1 P_1 + f_2 P_2 + f_3 P_3 =$
$g_0 + g_1 N_1 + g_2 N_2 + g_3 N_3$ and
$f_4 + f_5 P_1 + f_6 P_2 + f_7 P_3 =$
$g_4 + g_5 N_1 + g_6 N_2 + g_7 N_3$
then

$f_0 + f_1 P_1 + f_2 P_2 + f_3 P_3 +$
$f_4 P_4 + f_5 P_5 + f_6 P_6 + f_7 P_7 =$
$g_0 + (g_1 + g_7) N_1 +$
$(g_2 + g_6) N_2 + (g_3 + g_5) N_3 +$
$g_4 N_4 + g_5 N_5 + g_6 N_6 + g_7 N_7$.

Proof: e.g.,

$(\zeta+\zeta^{-1})^4(\zeta^3+\zeta^{-3})$
$=\zeta^7+\zeta^{-7}+\zeta^1+\zeta^{-1}$
so $P_4 N_3 = N_7 + N_1$. Q.E.D.

So size-8 conversion
from $1, P_1, P_2, \ldots, P_7$
to $1, N_1, N_2, \ldots, N_7$
can be done with
two size-4 conversions
and three additions.

Apply same idea recursively:
size-n conversion uses
$\le 1 + 0.5n(\lg n - 2)$ additions.

Inverse has same cost.

To multiply f, g on basis $N_1, N_2, \ldots, N_n$:

Convert to $1, P_1, \ldots, P_n$; cost $\approx 0.5n \lg n$, twice.

Polynomial product; $M(n+1)$.

Convert $1, P_1, \ldots, P_{2n}$ to $1, N_1, \ldots, N_{2n}$; cost $\approx n \lg n$.

Eliminate $N_{n+1}, \ldots, N_{2n}$ using $N_{2n+1-j} = N_j$; cost n.
Eliminate 1 using $1 + N_1 + \cdots + N_n = 0$; cost n.

Some new improvements:

1. For $1, P_1, \ldots, P_n$:
coefficient of 1 is 0.
Cost $M(n)$ instead of $M(n+1)$.

2. For $1, P_1, \ldots, P_{2n}$:
coefficients of $1, P_1$ are 0.
Reduces cost by $n+1$.

3. If mults share input,
reuse input conversion.
Reduces cost by $\approx 0.5n \lg n$.

4. If output is an input,
use different reduction strategy
to skip a first-half conversion.
Reduces cost by $\approx 0.5n \lg n$.

Can represent field element
using basis $P_1, \ldots, P_n$
for fast multiplication;
or basis $N_1, \ldots, N_n$
for fast multi-squarings;
or both.
Can vary this choice
across field-element variables.
Can also vary over time.

Approximate costs:
$P \to N$: $0.5n \lg n$.
$N \to P$: $0.5n \lg n$.
$P \times P \to N$: $M(n) + n \lg n$.
$P \times P \to P$: $M(n) + n \lg n$.
$N^{2^j} \to N$: 0.