# Optimizing linear maps modulo 2

(i.e.: fast xor sequences
for bitsliced software)

D. J. Bernstein
University of Illinois at Chicago

NSF ITR–0716498

Example: size-4 poly Karatsuba.

Start with size 2:

$F = F_0 + F_1 x$, $G = G_0 + G_1 x$,

$H_0 = F_0 G_0$, $H_2 = F_1 G_1$,

$H_1 = (F_0 + F_1)(G_0 + G_1) - H_0 - H_2$,

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

Substitute $x = t^2$ etc.:

$F = f_0 + f_1 t + f_2 t^2 + f_3 t^3$,

$G = f_0 + f_1 t + f_2 t^2 + f_3 t^3$,

$H_0 = (f_0 + f_1 t)(g_0 + g_1 t)$,

$H_2 = (f_2 + f_3 t)(g_2 + g_3 t)$,

$$\begin{aligned} H_1 = {} & (f_0 + f_2 + (f_1 + f_3)t) \cdot \\ & (g_0 + g_2 + (g_1 + g_3)t) \\ & - H_0 - H_2 \end{aligned}$$

$\Rightarrow FG = H_0 + H_1 t^2 + H_2 t^4$.

Initial linear computation:

$f_0 + f_2, f_1 + f_3, g_0 + g_2, g_1 + g_3$;
algebraic complexity 4.

Three size-2 mults producing

$H_0 = p_0 + p_1 t + p_2 t^2$;

$H_2 = q_0 + q_1 t + q_2 t^2$;

$H_0 + H_1 + H_2 = r_0 + r_1 t + r_2 t^2$.

Final linear reconstruction:

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

algebraic complexity 6;

$FG = H_0 + H_1 t^2 + H_2 t^4$,

algebraic complexity 2.

Let's look more closely
at the reconstruction:

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

Let's look more closely
at the reconstruction:

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

Can observe *manually*
that $p_2 - q_0$ is repeated.
See, e.g., 2000 Bernstein.

Some addition-chain algorithms
will *automatically*
find this speedup.

Consider, 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 would have
automatically found $p_2 \oplus q_0$
inside Karatsuba reconstruction.

Today's algorithm: "xor largest."
Start with the matrix mod 2
for the desired linear map.

$h_0$: 100000000
$h_1$: 010000000
$h_2$: 101100100
$h_3$: 010010010
$h_4$: 001101001
$h_5$: 000010000
$h_6$: 000001000

Each row has coefficients of
$p_0, p_1, p_2, q_0, q_1, q_2, r_0, r_1, r_2$.

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

100000000
010000000
001100100 ←
010010010
001101001
000010000
000001000

Recursively compute this,
and finish with one xor.

If two largest rows
don't have same first bit,
change largest row
by clearing first bit.

000000000 $\leftarrow$
010000000
001100100
010010010
001101001
000010000
000001000

Recursively compute this,
and finish with one xor
(often just a copy).

Continue in the same way:

100000000

010000000

101100100

010010010

001101001

000010000

000001000

(starting matrix again)

Continue in the same way:

100000000

010000000

001100100 $\leftarrow$

010010010

001101001

000010000

000001000

plus 1 xor.

Continue in the same way:

000000000 $\leftarrow$

010000000

001100100

010010010

001101001

000010000

000001000

plus 1 xor, 1 input load.

Continue in the same way:

000000000

010000000

001100100

000010010 $\leftarrow$

001101001

000010000

000001000

plus 2 xors, 1 input load.

Continue in the same way:

000000000

000000000 $\leftarrow$

001100100

000010010

001101001

000010000

000001000

plus 2 xors, 2 input loads.

Continue in the same way:

000000000
000000000
001100100
000010010
000001101 $\leftarrow$
000010000
000001000

plus 3 xors, 2 input loads.

Continue in the same way:

000000000
000000000
000100100 $\leftarrow$
000010010
000001101
000010000
000001000

plus 4 xors, 3 input loads.

Continue in the same way:

000000000
000000000
000000100 ←
000010010
000001101
000010000
000001000

plus 5 xors, 4 input loads.

Continue in the same way:

000000000
000000000
000000100
000000010 $\leftarrow$
000001101
000010000
000001000

plus 6 xors, 4 input loads.

Continue in the same way:

000000000
000000000
000000100
000000010
000001101
000000000 $\leftarrow$
000001000

plus 6 xors, 5 input loads.

Continue in the same way:

000000000
000000000
000000100
000000010
000000101 $\leftarrow$
000000000
000001000

plus 7 xors, 5 input loads.

Continue in the same way:

000000000
000000000
000000100
000000010
000000101
000000000
000000000 ←

plus 7 xors, 6 input loads.

Continue in the same way:

000000000
000000000
000000100
000000010
000000001 $\leftarrow$
000000000
000000000

plus 8 xors, 6 input loads.

Continue in the same way:

000000000

000000000

000000000 ←

000000010

000000001

000000000

000000000

plus 8 xors, 7 input loads.

Continue in the same way:

000000000
000000000
000000000
000000000 $\leftarrow$
000000001
000000000
000000000

plus 8 xors, 8 input loads.

Continue in the same way:

000000000

000000000

000000000

000000000

000000000 ←

000000000

000000000

plus 8 xors, 9 input loads.

Continue in the same way:

000000000
000000000
000000000
000000000
000000000 ←
000000000
000000000

plus 8 xors, 9 input loads.

“Is this supposed to be
an interesting algorithm?”

Another example:

000100000
000010000
100101100
010010010
001001101
000000010
000000001

Same matrix, but inputs
in a different order:
first $r$'s (used once each),
then $p$'s (used twice each),
then $q$'s (used twice each).

Another example:

000100000

000010000

000101100 $\leftarrow$

010010010

001001101

000000010

000000001

plus 1 xor, 1 input load.

Another example:

000100000

000010000

000101100

000010010 $\leftarrow$

001001101

000000010

000000001

plus 2 xors, 2 input loads.

Another example:

000100000

000010000

000101100

000010010

000001101 ←

000000010

000000001

plus 3 xors, 3 input loads.

Another example:

000100000

000010000

000001100 $\leftarrow$

000010010

000001101

000000010

000000001

plus 4 xors, 3 input loads.

Another example:

000000000 ←

000010000

000001100

000010010

000001101

000000010

000000001

plus 4 xors, 4 input loads.

Another example:

000000000

000010000

000001100

000000010 $\leftarrow$

000001101

000000010

000000001

plus 5 xors, 4 input loads.

Another example:

000000000

000000000 ←

000001100

000000010

000001101

000000010

000000001

plus 5 xors, 5 input loads.

Another example:

000000000
000000000
000001100
000000010
000000001 ←
000000010
000000001

plus 6 xors, 5 input loads.

Another example:

000000000

000000000

000000100 ←

000000010

000000001

000000010

000000001

plus 7 xors, 6 input loads.

Another example:

000000000
000000000
000000000 ←
000000010
000000001
000000010
000000001

plus 7 xors, 7 input loads.

Another example:

000000000
000000000
000000000
000000000 ←
000000001
000000010
000000001

plus 7 xors, 7 input loads.

Another example:

000000000
000000000
000000000
000000000
000000001
000000000 ←
000000001

plus 7 xors, 8 input loads.

Another example:

000000000
000000000
000000000
000000000
000000000 ←
000000000
000000001

plus 7 xors, 8 input loads.

Another example:

000000000
000000000
000000000
000000000
000000000
000000000
000000000 ←

plus 7 xors, 9 input loads.

Algorithm found the speedup.

Another example:

000000000
000000000
000000000
000000000
000000000
000000000
000000000 ←

plus 7 xors, 9 input loads.

Algorithm found the speedup.

Also has other useful features.

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.

Case study of benefits
produced by xor-largest:

131-bit conversion from
poly basis to normal basis.
"Random" $131 \times 131$ matrix.

On Cell ($\leq 1$ xor per cycle,
$128 - \epsilon$ registers) bitsliced
code took $\approx 9600$ cycles.

Output of xor-largest:
code with only 3380 xors
fitting into 132 registers.
Schwabe tuned asm for Cell:
$\approx 4000$ cycles.

Inspiration: 1989 Bos–Coster.

$000100000 = 32$

$000010000 = 16$

$100101100 = 300$

$010010010 = 146$

$001001101 = 77$

$000000010 = 2$

$000000001 = 1$

Goal: Compute $32x$, $16x$, $300x$, $146x$, $77x$, $2x$, $1x$.

Reduce largest row:

$000100000 = 32$

$000010000 = 16$

$010011010 = 154 \leftarrow$

$010010010 = 146$

$001001101 = 77$

$000000010 = 2$

$000000001 = 1$

*Integer* subtraction
of 146 from 300.

Reduce largest row:

$$000100000 = 32$$
$$000010000 = 16$$
$$000001000 = 8 \leftarrow$$
$$010010010 = 146$$
$$001001101 = 77$$
$$000000010 = 2$$
$$000000001 = 1$$

plus 2 additions.

Reduce largest row:

$$000100000 = 32$$
$$000010000 = 16$$
$$000001000 = 8$$
$$001000101 = 69 \leftarrow$$
$$001001101 = 77$$
$$000000010 = 2$$
$$000000001 = 1$$

plus 3 additions.

Reduce largest row:

$$000100000 = 32$$
$$000010000 = 16$$
$$000001000 = 8$$
$$001000101 = 69$$
$$000001000 = 8 \leftarrow$$
$$000000010 = 2$$
$$000000001 = 1$$

plus 4 additions.

Reduce largest row:

$$000100000 = 32$$
$$000010000 = 16$$
$$000001000 = 8$$
$$000100101 = 37 \leftarrow$$
$$000001000 = 8$$
$$000000010 = 2$$
$$000000001 = 1$$

plus 5 additions.

Reduce largest row:

$$000100000 = 32$$
$$000010000 = 16$$
$$000001000 = 8$$
$$000000101 = 5 \leftarrow$$
$$000001000 = 8$$
$$000000010 = 2$$
$$000000001 = 1$$

plus 6 additions.

Reduce largest row:

$000010000 = 16 \leftarrow$

$000010000 = 16$

$000001000 = 8$

$000000101 = 5$

$000001000 = 8$

$000000010 = 2$

$000000001 = 1$

plus 7 additions.

Reduce largest row:

$$000000000 = 0$$
$$000010000 = 16$$
$$000001000 = 8$$
$$000000101 = 5$$
$$000001000 = 8$$
$$000000010 = 2$$
$$000000001 = 1$$

plus 7 additions.

Reduce largest row:

$$000000000 = 0$$
$$000001000 = 8 \leftarrow$$
$$000001000 = 8$$
$$000000101 = 5$$
$$000001000 = 8$$
$$000000010 = 2$$
$$000000001 = 1$$

plus 8 additions.

Reduce largest row:

$$000000000 = 0$$
$$000000000 = 0 \leftarrow$$
$$000001000 = 8$$
$$000000101 = 5$$
$$000001000 = 8$$
$$000000010 = 2$$
$$000000001 = 1$$

plus 8 additions.

Reduce largest row:

$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0 \leftarrow$$
$$000000101 = 5$$
$$000001000 = 8$$
$$000000010 = 2$$
$$000000001 = 1$$

plus 8 additions.

Reduce largest row:

$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0$$
$$000000101 = 5$$
$$000000011 = 3 \leftarrow$$
$$000000010 = 2$$
$$000000001 = 1$$

plus 9 additions.

Reduce largest row:

$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0$$
$$000000010 = 2 \leftarrow$$
$$000000011 = 3$$
$$000000010 = 2$$
$$000000001 = 1$$

plus 10 additions.

Reduce largest row:

$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0$$
$$000000010 = 2$$
$$000000001 = 1 \leftarrow$$
$$000000010 = 2$$
$$000000001 = 1$$

plus 11 additions.

Reduce largest row:

$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0 \leftarrow$$
$$000000001 = 1$$
$$000000010 = 2$$
$$000000001 = 1$$

plus 11 additions.

Reduce largest row:

$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0$$
$$000000001 = 1$$
$$000000001 = 1 \leftarrow$$
$$000000001 = 1$$

plus 12 additions.

Reduce largest row:

$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0 \leftarrow$$
$$000000001 = 1$$
$$000000001 = 1$$

plus 12 additions.

Reduce largest row:

$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0 \leftarrow$$
$$000000001 = 1$$

plus 12 additions.

Reduce largest row:

$000000000 = 0$
$000000000 = 0$
$000000000 = 0$
$000000000 = 0$
$000000000 = 0$
$000000000 = 0$
$000000000 = 0 \leftarrow$

plus 12 additions.

Final addition chain: 1, 2, 3, 5, 8, 16, 32, 37, 69, 77, 146, 154, 300.

Short, no temporary storage, low two-operand complexity, etc.

Can imagine many other
mod-2 adaptations
of the Bos–Coster idea.

In reducing largest row:
Why use largest of
the remaining rows?
Why not minimize xor?

Out of first-bit-set rows:
Why do largest row first?
Why not start in middle,
or build Hamming tree?

Can reduce xors without
compromising regs etc.
I'm continuing to experiment.