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.:

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

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

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

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

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

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

zing linear maps modulo 2

ast xor sequences
sliced software)

Bernstein
sity of Illinois at Chicago

TR–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.:

$F = f_0 + f_1 t + f_2 t^2 + f_3 t^3$,
$G = f_0 + f_1 t + f_2 t^2 + f_3 t^3$,
$H_0 = (f_0 + f_1 t)(g_0 + g_1 t)$,
$H_2 = (f_2 + f_3 t)(g_2 + g_3 t)$,
$H_1 = (f_0 + f_2 + (f_1 + f_3)t) \cdot$
$(g_0 + g_2 + (g_1 + g_3)t)$
$- H_0 - H_2$
$\Rightarrow FG = H_0 + H_1 t^2 + H_2 t^4$.

Initial l
$f_0 + f_2$
algebra

Three s
$H_0 = p$
$H_2 = q$
$H_0 + H$

Final li
$H_1 = ($
(
(
algebra
$FG =$
algebra

r maps modulo 2

uences

vare)

ois at Chicago

98

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.:

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

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

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

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

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

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

Initial linear com

$f_0 + f_2, f_1 + f_3,$

algebraic comple

Three size-2 mul

$H_0 = p_0 + p_1 t +$

$H_2 = q_0 + q_1 t +$

$H_0 + H_1 + H_2 =$

Final linear recon

$H_1 = (r_0 - p_0 -$

$(r_1 - p_1 -$

$(r_2 - p_2 -$

algebraic comple

$FG = H_0 + H_1 t^2$

algebraic comple

odulo 2

cago

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.:

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

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

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

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

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

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

Initial linear computation:

$f_0 + f_2, f_1 + f_3, g_0 + g_2, g$

algebraic complexity 4.

Three size-2 mults produci

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

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

$H_0 + H_1 + H_2 = r_0 + r_1 t$

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.

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.:

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

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

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

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

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

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

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.

le: size-4 poly Karatsuba.

vith size 2:

$_0 + F_1 x$, $G = G_0 + G_1 x$,

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

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

$= H_0 + H_1 x + H_2 x^2$.

tute $x = t^2$ etc.:

$_0 + f_1 t + f_2 t^2 + f_3 t^3$,

$_0 + f_1 t + f_2 t^2 + f_3 t^3$,

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

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

$f_0 + f_2 + (f_1 + f_3)t) \cdot$

$g_0 + g_2 + (g_1 + g_3)t)$

$- H_0 - H_2$

$= H_0 + H_1 t^2 + H_2 t^4$.

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 lo

at the

$h_0 = p$

$h_1 = p$

$h_2 = p$

$h_3 = ($

$h_4 = ($

$h_5 = q$

$h_6 = q$

poly Karatsuba.

:

$G = G_0 + G_1 x$,

$= F_1 G_1$,

$G_0 + G_1) - H_0 - H_2$,

$H_1 x + H_2 x^2$.

^{2} etc.:

$f_2 t^2 + f_3 t^3$,

$f_2 t^2 + f_3 t^3$,

$(g_0 + g_1 t)$,

$(g_2 + g_3 t)$,

$(f_1 + f_3) t) \cdot$

$(g_1 + g_3) t)$

2

$H_1 t^2 + H_2 t^4$.

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 c

at the reconstruc

$h_0 = p_0$;

$h_1 = p_1$;

$h_2 = p_2 + (r_0 -$

$h_3 = (r_1 - p_1 -$

$h_4 = (r_2 - p_2 -$

$h_5 = q_1$;

$h_6 = q_2$.

tsuba.

$G_1 x$,

$H_0 - H_2$,

t^2.

^{3},

^{3},

,

,

$t) \cdot$

$t)$

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:

$h_0 = p_0$;

$h_1 = p_1$;

$h_2 = p_2 + (r_0 - p_0 - q_0)$;

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

$h_4 = (r_2 - p_2 - q_2) + q_0$;

$h_5 = q_1$;

$h_6 = q_2$.

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:

$h_0 = p_0$;

$h_1 = p_1$;

$h_2 = p_2 + (r_0 - p_0 - q_0)$;

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

$h_4 = (r_2 - p_2 - q_2) + q_0$;

$h_5 = q_1$;

$h_6 = q_2$.

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:

$h_0 = p_0$;

$h_1 = p_1$;

$h_2 = p_2 + (r_0 - p_0 - q_0)$;

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

$h_4 = (r_2 - p_2 - q_2) + q_0$;

$h_5 = q_1$;

$h_6 = q_2$.

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

inear computation:

$_2, f_1 + f_3, g_0 + g_2, g_1 + g_3$;

ic complexity 4.

size-2 mults producing

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

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

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

near reconstruction:

$(r_0 - p_0 - q_0) +$

$(r_1 - p_1 - q_1)t +$

$(r_2 - p_2 - q_2)t^2$,

ic complexity 6;

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

ic complexity 2.

Let's look more closely

at the reconstruction:

$h_0 = p_0$;

$h_1 = p_1$;

$h_2 = p_2 + (r_0 - p_0 - q_0)$;

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

$h_4 = (r_2 - p_2 - q_2) + q_0$;

$h_5 = q_1$;

$h_6 = q_2$.

Can observe *manually*

that $p_2 - q_0$ is repeated.

See, e.g., 2000 Bernstein.

Some a

will *au*

find th

Consid

CSE al

- find i
  with
- comp
- simpl
- repea

This al

automa

inside

putation:

$g_0 + g_2, g_1 + g_3$;

xity 4.

ts producing

$+ p_2t^2$;

q_2t^2;

$= r_0 + r_1t + r_2t^2$.

nstruction:

$q_0) +$

$q_1)t +$

$q_2)t^2$,

xity 6;

$^2 + H_2t^4$,

xity 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-ch

will *automatically*

find this speedup

Consider, e.g., gr

CSE algorithm fr

- find input pair
  with most popu
- compute $i_0 \oplus i$
- simplify using i
- repeat.

This algorithm w

automatically fou

inside Karatsuba

$_1 + g_3$;

ng

$+ r_2 t^2$.

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 algori will *automatically* find this speedup.

Consider, e.g., greedy addi CSE algorithm from 1997

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

This algorithm would have automatically found $p_2 \oplus q$ inside Karatsuba reconstru

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.

ook more closely
reconstruction:

$_0$;

$_1$;

$_2 + (r_0 - p_0 - q_0)$;

$r_1 - p_1 - q_1)$;

$r_2 - p_2 - q_2) + q_0$;

1;

2.

oserve *manually*
$_2 - q_0$ is repeated.
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'
Start w
for the

h_0: 10
h_1: 01
h_2: 10
h_3: 01
h_4: 00
h_5: 00
h_6: 00

Each ro
$p_0, p_1,$

closely

ction:

$p_0 - q_0)$;

$q_1)$;

$q_2) + q_0$;

nually

epeated.

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 algorith

Start with the m

for the desired li

h_0: 100000000

h_1: 010000000

h_2: 101100100

h_3: 010010010

h_4: 001101001

h_5: 000010000

h_6: 000001000

Each row has coe

$p_0, p_1, p_2, q_0, q_1,$

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 la

Start with the matrix mod

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 o

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

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

addition-chain algorithms
tomatically
is speedup.

er, e.g., greedy additive
gorithm from 1997 Paar:
input pair i_0, i_1
most popular $i_0 \oplus i_1$;
oute $i_0 \oplus i_1$;
lify using $i_0 \oplus i_1$;
at.

gorithm would have
atically found $p_2 \oplus q_0$
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
$p_0, p_1, p_2, q_0, q_1, q_2, r_0, r_1, r_2$.

Replac
by its x
second-

100000
010000
001100
010010
001101
000010
000001

Recursi
and fin

nain algorithms
y
.

reedy additive
rom 1997 Paar:
i_0, i_1
ular $i_0 \oplus i_1$;
i_1;
$i_0 \oplus i_1$;

vould have
und $p_2 \oplus q_0$
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 r
by its xor with
second-largest ro

100000000
010000000
001100100 $\leftarrow$
010010010
001101001
000010000
000001000

Recursively comp
and finish with o

thms

tive

Paar:

$_1$;

$_0$

ction.

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 $\leftarrow$

010010010

001101001

000010000

000001000

Recursively compute this, and finish with one xor.

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 $\leftarrow$
010010010
001101001
000010000
000001000

Recursively compute this,
and finish with one xor.

s algorithm: "xor largest."
vith the matrix mod 2
desired linear map.

0000000
0000000
1100100
0010010
1101001
0010000
0001000

ow has coefficients of
$p_2, q_0, q_1, q_2, r_0, r_1, r_2$.

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

100000000
010000000
001100100 $\leftarrow$
010010010
001101001
000010000
000001000

Recursively compute this,
and finish with one xor.

If two l
don't h
change
by clea

00000C
01000C
00110C
01001C
001101
00001C
000001

Recursi
and fin
(often

m: “xor largest.”
atrix mod 2
near map.

efficients of
q_2, r_0, r_1, r_2.

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

100000000
010000000
001100100 $\leftarrow$
010010010
001101001
000010000
000001000

Recursively compute this,
and finish with one xor.

If two largest row
don’t have same
change largest ro
by clearing first b

000000000 $\leftarrow$
010000000
001100100
010010010
001101001
000010000
000001000

Recursively comp
and finish with o
(often just a cop

rgest."

2

of

r_2.

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

100000000
010000000
001100100 $\leftarrow$
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).

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

100000000
010000000
001100100 $\leftarrow$
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).

e largest row
xor with
-largest row.

0000
0000
0100 ←
0010
1001
0000
1000

ively compute this,
ish with one xor.

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

000000000 ←
010000000
001100100
010010010
001101001
000010000
000001000

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

Contin

100000
010000
101100
010010
001101
000010
000001

(startin

ow

w.

pute this,

ne xor.

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

000000000 ←
010000000
001100100
010010010
001101001
000010000
000001000

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

Continue in the s

100000000
010000000
101100100
010010010
001101001
000010000
000001000

(starting matrix a

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

000000000 ←
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)

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)

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
001100100 $\leftarrow$
010010010
001101001
000010000
000001000

plus 1 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:

000000000 $\leftarrow$
010000000
001100100
010010010
001101001
000010000
000001000

plus 1 xor, 1 input load.

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

000000000 ←
010000000
001100100
010010010
001101001
000010000
000001000

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

Continue in the same way:

000000000
010000000
001100100
000010010 ←
001101001
000010000
000001000

plus 2 xors, 1 input load.

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:

000000000
000000000 $\leftarrow$
001100100
000010010
001101001
000010000
000001000

plus 2 xors, 2 input loads.

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:

000000000
000000000
001100100
000010010
000001101 $\leftarrow$
000010000
000001000

plus 3 xors, 2 input loads.

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:

000000000
000000000
000100100 $\leftarrow$
000010010
000001101
000010000
000001000

plus 4 xors, 3 input loads.

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:

000000000
000000000
000000100 $\leftarrow$
000010010
000001101
000010000
000001000

plus 5 xors, 4 input loads.

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:

000000000
000000000
000000100
000000010 $\leftarrow$
000001101
000010000
000001000

plus 6 xors, 4 input loads.

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:

000000000
000000000
000000100
000000010
000001101
000000000 $\leftarrow$
000001000

plus 6 xors, 5 input loads.

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:

000000000
000000000
000000100
000000010
000000101 $\leftarrow$
000000000
000001000

plus 7 xors, 5 input loads.

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

000000000 ←
010000000
001100100
010010010
001101001
000010000
000001000

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

Continue in the same way:

000000000
000000000
000000100
000000010
000000101
000000000
000000000 ←

plus 7 xors, 6 input loads.

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:

000000000
000000000
000000100
000000010
000000001 $\leftarrow$
000000000
000000000

plus 8 xors, 6 input loads.

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

000000000 ←
010000000
001100100
010010010
001101001
000010000
000001000

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

Continue in the same way:

000000000
000000000
000000000 ←
000000010
000000001
000000000
000000000

plus 8 xors, 7 input loads.

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

000000000 ←
010000000
001100100
010010010
001101001
000010000
000001000

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

Continue in the same way:

000000000
000000000
000000000
000000000 ←
000000001
000000000
000000000

plus 8 xors, 8 input loads.

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:

000000000
000000000
000000000
000000000
000000000 $\leftarrow$
000000000
000000000

plus 8 xors, 9 input loads.

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:

000000000
000000000
000000000
000000000
000000000 $\leftarrow$
000000000
000000000

plus 8 xors, 9 input loads.

"Is this supposed to be
an interesting algorithm?"

largest rows
have same first bit,
e largest row
ring first bit.

0000 ←
0000
0100
0010
1001
0000
1000

ively compute this,
ish with one xor
just a copy).

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?"

Anothe

000100
000010
100101
010010
001001
000000
000000

Same r
in a dif
first r's
then p'
then q'

ws
first bit,
ow
bit.

pute this,
ne xor
y).

Continue in the same way:

000000000
000000000
000000000
000000000
000000000 $\leftarrow$
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, bu
in a different ord
first r's (used on
then p's (used tw
then q's (used tw

Continue in the same way:

000000000
000000000
000000000
000000000
000000000 $\leftarrow$
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).

Continue in the same way:

000000000
000000000
000000000
000000000
000000000 $\leftarrow$
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).

Continue in the same way:

000000000

000000000

000000000

000000000

000000000 $\leftarrow$

000000000

000000000

plus 8 xors, 9 input loads.

"Is this supposed to be
an interesting algorithm?"

Another example:

000100000

000010000

000101100 $\leftarrow$

010010010

001001101

000000010

000000001

plus 1 xor, 1 input load.

Continue in the same way:

000000000

000000000

000000000

000000000

000000000 $\leftarrow$

000000000

000000000

plus 8 xors, 9 input loads.

"Is this supposed to be
an interesting algorithm?"

Another example:

000100000

000010000

000101100

000010010 $\leftarrow$

001001101

000000010

000000001

plus 2 xors, 2 input loads.

Continue in the same way:

000000000
000000000
000000000
000000000
000000000 $\leftarrow$
000000000
000000000

plus 8 xors, 9 input loads.

"Is this supposed to be
an interesting algorithm?"

Another example:

000100000
000010000
000101100
000010010
000001101 $\leftarrow$
000000010
000000001

plus 3 xors, 3 input loads.

Continue in the same way:

000000000

000000000

000000000

000000000

000000000 $\leftarrow$

000000000

000000000

plus 8 xors, 9 input loads.

"Is this supposed to be an interesting algorithm?"

Another example:

000100000

000010000

000001100 $\leftarrow$

000010010

000001101

000000010

000000001

plus 4 xors, 3 input loads.

Continue in the same way:

000000000

000000000

000000000

000000000

000000000 $\leftarrow$

000000000

000000000

plus 8 xors, 9 input loads.

"Is this supposed to be
an interesting algorithm?"

Another example:

000000000 $\leftarrow$

000010000

000001100

000010010

000001101

000000010

000000001

plus 4 xors, 4 input loads.

Continue in the same way:

000000000

000000000

000000000

000000000

000000000 $\leftarrow$

000000000

000000000

plus 8 xors, 9 input loads.

"Is this supposed to be an interesting algorithm?"

Another example:

000000000

000010000

000001100

000000010 $\leftarrow$

000001101

000000010

000000001

plus 5 xors, 4 input loads.

Continue in the same way:

000000000

000000000

000000000

000000000

000000000 $\leftarrow$

000000000

000000000

plus 8 xors, 9 input loads.

"Is this supposed to be
an interesting algorithm?"

Another example:

000000000

000000000 $\leftarrow$

000001100

000000010

000001101

000000010

000000001

plus 5 xors, 5 input loads.

Continue in the same way:

000000000
000000000
000000000
000000000
000000000 $\leftarrow$
000000000
000000000

plus 8 xors, 9 input loads.

"Is this supposed to be
an interesting algorithm?"

Another example:

000000000
000000000
000001100
000000010
000000001 $\leftarrow$
000000010
000000001

plus 6 xors, 5 input loads.

Continue in the same way:

000000000

000000000

000000000

000000000

000000000 $\leftarrow$

000000000

000000000

plus 8 xors, 9 input loads.

"Is this supposed to be
an interesting algorithm?"

Another example:

000000000

000000000

000000100 $\leftarrow$

000000010

000000001

000000010

000000001

plus 7 xors, 6 input loads.

Continue in the same way:

000000000
000000000
000000000
000000000
000000000 $\leftarrow$
000000000
000000000

plus 8 xors, 9 input loads.

"Is this supposed to be
an interesting algorithm?"

Another example:

000000000
000000000
000000000 $\leftarrow$
000000010
000000001
000000010
000000001

plus 7 xors, 7 input loads.

Continue in the same way:

000000000

000000000

000000000

000000000

000000000 $\leftarrow$

000000000

000000000

plus 8 xors, 9 input loads.

“Is this supposed to be
an interesting algorithm?”

Another example:

000000000

000000000

000000000

000000000 $\leftarrow$

000000001

000000010

000000001

plus 7 xors, 7 input loads.

Continue in the same way:

000000000

000000000

000000000

000000000

000000000 $\leftarrow$

000000000

000000000

plus 8 xors, 9 input loads.

"Is this supposed to be an interesting algorithm?"

Another example:

000000000

000000000

000000000

000000000

000000001

000000000 $\leftarrow$

000000001

plus 7 xors, 8 input loads.

Continue in the same way:

000000000
000000000
000000000
000000000
000000000 $\leftarrow$
000000000
000000000

plus 8 xors, 9 input loads.

"Is this supposed to be
an interesting algorithm?"

Another example:

000000000
000000000
000000000
000000000
000000000 $\leftarrow$
000000000
000000001

plus 7 xors, 8 input loads.

Continue in the same way:

000000000

000000000

000000000

000000000

000000000 $\leftarrow$

000000000

000000000

plus 8 xors, 9 input loads.

"Is this supposed to be
an interesting algorithm?"

Another example:

000000000

000000000

000000000

000000000

000000000

000000000

000000000 $\leftarrow$

plus 7 xors, 9 input loads.

Algorithm found the speedup.

Continue in the same way:

000000000
000000000
000000000
000000000
000000000 $\leftarrow$
000000000
000000000

plus 8 xors, 9 input loads.

"Is this supposed to be an interesting algorithm?"

Another example:

000000000
000000000
000000000
000000000
000000000
000000000
000000000 $\leftarrow$

plus 7 xors, 9 input loads.

Algorithm found the speedup.

Also has other useful features.

ue in the same way:

0000
0000
0000
0000
0000 $\leftarrow$
0000
0000

xors, 9 input loads.

s supposed to be
resting algorithm?”

Another example:

000000000
000000000
000000000
000000000
000000000
000000000
000000000 $\leftarrow$

plus 7 xors, 9 input loads.

Algorithm found the speedup.

Also has other useful features.

Memor
Algorit
to the
No tem

n inpu
total 2
with 0

Or $n +$
with n
each in

Or n re
with n
if platf

same way:

out loads.

l to be

gorithm?"

Another example:

000000000

000000000

000000000

000000000

000000000

000000000

000000000 $\leftarrow$

plus 7 xors, 9 input loads.

Algorithm found the speedup.

Also has other useful features.

Memory friendlin

Algorithm writes

to the output reg

No temporary sto

n inputs, n outp

total $2n$ registers

with 0 loads, 0 st

Or $n+1$ register

with n loads, 0 s

each input is rea

Or n registers

with n loads, 0 s

if platform has lo

Another example:

000000000
000000000
000000000
000000000
000000000
000000000
000000000 $\leftarrow$

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 onc

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

Another example:

000000000
000000000
000000000
000000000
000000000
000000000
000000000 $\leftarrow$

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.

er example:

0000

0000

0000

0000

0000

0000

0000 $\leftarrow$

xors, 9 input loads.

hm found the speedup.

as 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-op
Platfor
but wit
uses or

Naive c
$n+1$ r
but usu

Input p
(e.g., 1
somew
somew

Greedy
somew
many r

e:

out loads.

the speedup.

seful 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 frie
Platform with a
but without $a \leftarrow$
uses only n extra

Naive column sw
$n+1$ registers, n
but usually many

Input partitioning
(e.g., 1956 Lupar
somewhat more x
somewhat more r

Greedy additive C
somewhat fewer
many more copie

up.

res.

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 u
$n+1$ registers, n loads,
but usually many more xor

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

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

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.

y friendliness:
hm writes only
output registers.
nporary storage.

ts, n outputs:
n registers
loads, 0 stores.

$+ 1$ registers
loads, 0 stores:
put is read only once.

egisters
loads, 0 stores,
orm 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
average

The xo
$\approx mn/$
n copie

ess:
only
gisters.
orage.

uts:
s
tores.

rs
stores:
d only once.

stores,
oad-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 an
average $n \times m$ n

The xor-largest a
$\approx mn/\lg n$ two-
n copies; m load

ce.

sn.

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 outpu
average $n \times m$ matrix:

The xor-largest algorithm u
$\approx mn/\lg n$ two-operand x
n copies; m loads; $n + 1$ r

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.

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.

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.

perand friendliness:
m with $a \leftarrow a \oplus b$
thout $a \leftarrow b \oplus c$
nly n extra copies.

column sweep also uses
registers, n loads,
ually many more xors.

partitioning
956 Lupanov) uses
hat more xors, copies;
hat more registers.

additive CSE uses
hat fewer xors but
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.

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 st
produc

131-bit
poly ba
"Rando

On Cel
128 −
code to

Output
code w
fitting
Schwab
≈ 4000

endliness:

$\leftarrow a \oplus b$

$b \oplus c$

a copies.

eep also uses

n loads,

y more xors.

g

nov) uses

xors, copies;

registers.

CSE uses

xors but

es, 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.

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 be

produced by xor-

131-bit conversio

poly basis to nor

"Random" 131 ×

On Cell ($\leq$ 1 xor

128 − ϵ registers

code took $\approx$ 960

Output of xor-lar

code with only 3

fitting into 132 r

Schwabe tuned a

$\approx$ 4000 cycles.

ses

s.

es;

s.

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" 131×131 matr

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

Output of xor-largest: code with only 3380 xors fitting into 132 registers. Schwabe tuned asm for Ce ≈ 4000 cycles.

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" 131×131 matrix.

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

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

inputs and n outputs,
e $n \times m$ matrix:

or-largest algorithm uses
/ lg n two-operand xors;
es; m loads; $n+1$ regs.

ger's algorithm uses
/ lg mn three-operand xors
ems to need many regs.

ger proved that
ebraic complexity was
ptimal for most matrices
st without mod 2),
ln't consider regs,
erand complexity, etc.

Case study of benefits
produced by xor-largest:

131-bit conversion from
poly basis to normal basis.
"Random" 131×131 matrix.

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

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

Inspira

000100
000010
100101
010010
001001
000000
000000

Goal: C
$300x$, 1

d n outputs,
matrix:

lgorithm uses
operand xors;
ls; $n + 1$ regs.

rithm uses
ree-operand xors
ed many regs.

d that
nplexity was
most matrices
mod 2),
er regs,
nplexity, etc.

Case study of benefits produced by xor-largest:

131-bit conversion from poly basis to normal basis. "Random" 131×131 matrix.

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

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

Inspiration: 1989

$000100000 = 32$
$000010000 = 16$
$100101100 = 300$
$010010010 = 146$
$001001101 = 77$
$000000010 = 2$
$000000001 = 1$

Goal: Compute 3
$300x$, $146x$, $77x$

ts,

uses
ors;
egs.

.
nd xors
egs.

as
rices

tc.

Case study of benefits
produced by xor-largest:

131-bit conversion from
poly basis to normal basis.
“Random” 131×131 matrix.

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

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

Inspiration: 1989 Bos–Cos

$000100000 = 32$
$000010000 = 16$
$100101100 = 300$
$010010010 = 146$
$001001101 = 77$
$000000010 = 2$
$000000001 = 1$

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

Case study of benefits produced by xor-largest:

131-bit conversion from poly basis to normal basis. "Random" 131×131 matrix.

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

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

Inspiration: 1989 Bos–Coster.

$000100000 = 32$

$000010000 = 16$

$100101100 = 300$

$010010010 = 146$

$001001101 = 77$

$000000010 = 2$

$000000001 = 1$

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

tudy of benefits
ed by xor-largest:

t conversion from
asis to normal basis.
om" 131×131 matrix.

l (≤ 1 xor per cycle,
ε registers) bitsliced
ook ≈ 9600 cycles.

t of xor-largest:
vith only 3380 xors
into 132 registers.
be tuned asm for Cell:
0 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

00010
00001
01001
01001
00100
00000
00000

Integer
of 146

nefits

largest:

on from

mal basis.

< 131 matrix.

per cycle,

) bitsliced

0 cycles.

rgest:

380 xors

egisters.

nsm for Cell:

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 ro

$000100000 = 32$

$000010000 = 16$

$010011010 = 154$

$010010010 = 146$

$001001101 = 77$

$000000010 = 2$

$000000001 = 1$

Integer subtracti of 146 from 300.

rix.

,

ll:

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.

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.

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:

$000100000 = 32$

$000010000 = 16$

$000001000 = 8 \leftarrow$

$010010010 = 146$

$001001101 = 77$

$000000010 = 2$

$000000001 = 1$

plus 2 additions.

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:

$000100000 = 32$

$000010000 = 16$

$000001000 = 8$

$001000101 = 69 \leftarrow$

$001001101 = 77$

$000000010 = 2$

$000000001 = 1$

plus 3 additions.

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:

$000100000 = 32$

$000010000 = 16$

$000001000 = 8$

$001000101 = 69$

$000001000 = 8 \leftarrow$

$000000010 = 2$

$000000001 = 1$

plus 4 additions.

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:

$000100000 = 32$

$000010000 = 16$

$000001000 = 8$

$000100101 = 37 \leftarrow$

$000001000 = 8$

$000000010 = 2$

$000000001 = 1$

plus 5 additions.

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:

$000100000 = 32$

$000010000 = 16$

$000001000 = 8$

$000000101 = 5 \leftarrow$

$000001000 = 8$

$000000010 = 2$

$000000001 = 1$

plus 6 additions.

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:

$000010000 = 16 \leftarrow$

$000010000 = 16$

$000001000 = 8$

$000000101 = 5$

$000001000 = 8$

$000000010 = 2$

$000000001 = 1$

plus 7 additions.

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:

$000000000 = 0$

$000010000 = 16$

$000001000 = 8$

$000000101 = 5$

$000001000 = 8$

$000000010 = 2$

$000000001 = 1$

plus 7 additions.

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:

$000000000 = 0$

$000001000 = 8 \leftarrow$

$000001000 = 8$

$000000101 = 5$

$000001000 = 8$

$000000010 = 2$

$000000001 = 1$

plus 8 additions.

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:

$000000000 = 0$

$000000000 = 0 \leftarrow$

$000001000 = 8$

$000000101 = 5$

$000001000 = 8$

$000000010 = 2$

$000000001 = 1$

plus 8 additions.

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:

$000000000 = 0$

$000000000 = 0$

$000000000 = 0 \leftarrow$

$000000101 = 5$

$000001000 = 8$

$000000010 = 2$

$000000001 = 1$

plus 8 additions.

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:

$000000000 = 0$

$000000000 = 0$

$000000000 = 0$

$000000101 = 5$

$000000011 = 3 \leftarrow$

$000000010 = 2$

$000000001 = 1$

plus 9 additions.

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:

$000000000 = 0$

$000000000 = 0$

$000000000 = 0$

$000000010 = 2 \leftarrow$

$000000011 = 3$

$000000010 = 2$

$000000001 = 1$

plus 10 additions.

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:

$000000000 = 0$

$000000000 = 0$

$000000000 = 0$

$000000010 = 2$

$000000001 = 1 \leftarrow$

$000000010 = 2$

$000000001 = 1$

plus 11 additions.

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:

$000000000 = 0$

$000000000 = 0$

$000000000 = 0$

$000000000 = 0 \leftarrow$

$000000001 = 1$

$000000010 = 2$

$000000001 = 1$

plus 11 additions.

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:

$000000000 = 0$

$000000000 = 0$

$000000000 = 0$

$000000000 = 0$

$000000001 = 1$

$000000001 = 1 \leftarrow$

$000000001 = 1$

plus 12 additions.

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:

$000000000 = 0$

$000000000 = 0$

$000000000 = 0$

$000000000 = 0$

$000000000 = 0 \leftarrow$

$000000001 = 1$

$000000001 = 1$

plus 12 additions.

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:

$000000000 = 0$

$000000000 = 0$

$000000000 = 0$

$000000000 = 0$

$000000000 = 0$

$000000000 = 0 \leftarrow$

$000000001 = 1$

plus 12 additions.

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:

$000000000 = 0$

$000000000 = 0$

$000000000 = 0$

$000000000 = 0$

$000000000 = 0$

$000000000 = 0$

$000000000 = 0 \leftarrow$

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.

tion: 1989 Bos–Coster.

$0000 = 32$
$0000 = 16$
$.100 = 300$
$0010 = 146$
$.101 = 77$
$0010 = 2$
$0001 = 1$

Compute $32x$, $16x$,
$146x$, $77x$, $2x$, $1x$.

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 im
mod-2
of the

In redu
Why us
the ren
Why no

Out of
Why do
Why no
or build

Can red
compro
I'm con

Bos–Coster.

0

6

$32x$, $16x$,

, $2x$, $1x$.

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 mar
mod-2 adaptatio
of the Bos–Coste

In reducing large
Why use largest
the remaining ro
Why not minimiz

Out of first-bit-se
Why do largest r
Why not start in
or build Hammin

Can reduce xors
compromising reg
I'm continuing to

ter.

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 experime

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.