

# High-Performance Meataxe

Edinburgh July 2009

R. A. Parker

# Acknowledgements

- Thanks to Derek, Steve and Colva for organizing this conference and allowing me to talk
- To Steve for encouraging me to take a fresh look at meataxe performance 3-4 years ago
- To Beth Holmes (and Steve (yet) again) for collaboration in this work.
- But much of this is as-yet untried!

For matrix multiplication,  
like any binary function.

- You do as much work as you can that depends on 1 input
- And as little as possible that depends on both inputs.
- Like proving conjugacy of one element
- I think this conference is all about work on the isomorphism of . . . one group!
- Or work on multiplying . . . one matrix!





# Why reformat?

- To take advantage of “grease”
- To enlarge the set of possible algorithms.
- So that access to the data is faster for the computer.

# $A = B \times C$ The different formats.

- **A** – the format the answer comes out in
  - Suitable for accumulating into.
- **B** – the format matrix B should be in
  - Just a set of indexes into the grease table.
- **C** – the format matrix C should be in
  - **C1** Whole matrix ready to build grease table.
  - **C2** Partial matrix fully populated grease table.
- **D** – the format in the files.
  - Same as for last 20 years.

# Grease matrix $C$

- Format  $\underline{C}$  (at grease level  $g$ ) consists of all possible linear combinations of each set of  $g$  rows.
- Then, to add in any linear combination of rows of  $C$ , instead of  $g$  multiplies and  $g$  adds, just one add is needed.

# Example - Grease level 2 over F3.

```

.. 0 2 ..      .....
.. 1 0 .. D   1 2 2 0 0 2 1 2 1 1 1 ... D
.. 0 1 ..      2 0 1 1 0 1 2 1 1 0 2 ...
.. 1 1 ..      .....
== B == x ===== C =====

```

```

. 2           2 0 1 1 0 1 2 1 1 0 2 .....
. 3 B       1 0 2 2 0 2 1 2 2 0 1 ..... C2
. 1           1 2 2 0 0 2 1 2 1 1 1 .....
. 4           0 2 0 1 0 0 0 0 2 1 0 .....

```

# Grease level can be a rational number

- e.g. mod 1949 should use grease  $1/2$
- Store  $v, 2v, 3v, \dots, 45v$  ( $43 \times 46 > 1949$ )
- Also  $46v, 2.46v, 3.46v \dots, 42 \times 46v$
- So to add in  $131.v$  you add in
- $2.46v$  and  $39v$  ( $2 \times 46 + 39 = 131$ )
- Two adds is still much faster than a multiply (and reduction)
- Real gain is you reduced e.g.  $7.46v$  first.

# Wider Range of algorithms

- Example – **the field of order 32.**
- **D** (file format) has one entry per byte, using 5 bits.
- For **C** take 64 [actually word length] entries and put them into 5 words (p,q,r,s,t) so that, for example, the fifth field entry is bit 5 of each of p, q, r, s and t.
- (for **C2**, perhaps make 35 non-zero linear combinations of p,q,r,s and t (31, but four of them twice)
- For **B**, hold where to start, adding five consecutive rows of **C2** in to the five rows of **A**. Hold five zero rows somewhere too – faster to add in zero than decide not to
- **A** looks muck like **C** – at least as far as the addition and scalar multiplication is concerned.

# Wider range of algorithms

- Example – **the field of order 3.**
- Format **A** and **C** – need to store each field entry in corresponding bits of two words so that + (exclusive or) and & (logical and) can be done on a whole word at once.
- 7 ops – multiply in F4 to add in F3.
- But – can do it in 6 ops. . . found by looking at all possible “programs”!
- Notice **A** and **C** *must* be different formats.

# Adding mod 3

$R + C = S$        $R=(r1,r2)$     $C=(c1,c2)$     $S=(s1,s2)$   
 $x=r1+c2, y=r2+c1$     $z=x+c1$     $t=y+c2$     $s1=y\&z$     $s2=x\&t$

<b><i>r1</i></b>	<b><i>r2</i></b>	<b><i>c1</i></b>	<b><i>c2</i></b>	<b><i>x</i></b>	<b><i>y</i></b>	<b><i>z</i></b>	<b><i>t</i></b>	<b><i>s1</i></b>	<b><i>s2</i></b>	<b><i>R+C=S</i></b>
0	0	1	1	1	1	0	0	0	0	0+0=0
0	0	0	1	1	0	1	1	0	1	0+1=1
0	0	1	0	0	1	1	1	1	0	0+2=2
0	1	1	1	1	0	0	1	0	1	1+0=1
0	1	0	1	1	1	1	0	1	0	1+1=2
0	1	1	0	0	0	1	0	0	0	1+2=0
1	0	1	1	0	1	1	0	1	0	2+0=2
1	0	0	1	0	0	0	1	0	0	2+1=0
1	0	1	0	1	1	0	1	0	1	2+2=1

# Adding – moderate primes

<u>P</u>	<u>Bits</u>	<u>/64</u>	<u>Adds</u>	<u>Reduction</u>
5	7	9	15	64=4
7	8	8	21	128=2
11	9	7	25	256=1+2
13	10	6	41	512=1+4
17	10	6	31	512=2

Last row means mod 17, each entry takes 10 bits so you store 6 entries in one 64-bit word. You can add 31 times for each reduction, and to reduce, you “and” with 512, “xor”, shift right 8 places and “add”.

# Mod 3 is faster the first way

- Using 6 bits,  $32=2 \bmod 3$  adds 10 entries at a time and needs 4 operations every 15 adds so does 150 adds in 19 operations.
- Other way does 64 adds in 6 operations.
- Quite close, really.
- first way is also much tighter on (cache) memory and can negate by swapping words, so is in fact quite a bit faster.

# Mod 5

- If you can add mod 5 in 8 operations, that would be good. Should probably look.
- But I'm not really interested in adding mod 5 in 9 operations because . . .
- If there were, it would be about the same speed as adding in 7 bits.
- $135/19 = 7.10523$
- $64/9 = 7.11111$

# Mod 7

- 13 operations is easy – multiply in GF8
- $64/13 = 4.92$
- Addition in a byte is much faster
- $168/25 = 6.72$

## Moral of the story

Computers have good electronics for addition, so use it.

# Try to avoid reduction

- For example 1949 use 16 bits per entry
- Usually just add the next number in
  - SSE-2 can do 8 of these in 1 operation
- Every 15 adds (say) reduce it *a bit*
- “and” with 32768, shift right 15 places, multiply by 1584 and subtract.
  - SSE-2 can do these operations 8 at a time too
- Only at the very end do you bother to get the answer right by doing a (slow) %1949.

$23^5$  ??

- Grease to level “1/5”, - all 23 multiples.
- So needs five adds per entry
- Should do one matrix entry every 5.2 cycles
- Or about 2 seconds for 1000 x 1000. *in theory*
- But my defined format is  $a+23b+529c \dots$
- And getting to and from that is slow 😞

# Faster Access to the data

- The grease table **C2** is accessed randomly, so it is quite important that it fits into L2 cache.
- This usually implies cutting a large matrix C up in both directions so that the “little square” of C can be heavily greased, used and discarded.
- A and B are then stored (as parts of rows and indexes into the grease table respectively) so that all the data for each such little square is together and accessed sequentially.

# Example C2 – mod 2

- Chop matrix C up into 256 x 512 “squares”
- Row chunk is 64 bytes
- Grease level 8
- 32 “slices” of 8->256 rows – 524,768 in L2
- So chunk add is perhaps 8 x 64 bit Xor
  - Row being added into resides in L1 cache
  - Quite a bit of work per fetch of a B index and Read and Write-back of A chunk

# Meataxe matrices are not dense!

- Naively implemented, all this stuff was not much faster than the old methods (which did not chop the rows up).
- The old methods took advantage of the large number of zeros there often are in matrix B
- We can do that if a whole load of consecutive B-entries are zero
- Or if a whole Square of C2 is zero for that matter.
- Then the new methods are indeed much faster.

# HPMI

- High Performance Meataxe Interface
  - Not properly implemented yet.
  - Beth had a go at quite a bit of it.
- Run time, first job is to decide on a strategy (grease level, sizes of chunks etc.)
- Read matrix B then convert into format **B**
- Read matrix C then convert into format **C**
- Do  $A = B \times C$  on the most suitable formats
- Convert format **A** back to **D** and write out.

# Useful even for small matrices

- Consider format **B** (respectively format **C**) as a format suitable for multiplying on the right (respectively on the left)
- Suppose you wanted to make 100,000,000 matrices . . .
- Would 10,000  $B_i$  x 10,000  $C_j$  do?
- Convert all  $B_i$  into a format **B** first
- Then grease each  $C_j$  in turn (i.e. make **C2**)

# “Multiply and Add”

$$A += B \times C$$

- This is the performance critical operation of the meataxe for large matrices.
- Multiplication can obviously use it ( $A=0$ ).
- Gaussian elimination can also utilize this operation for its performance-critical parts
  - Must use the proper echelon form

# Layers (in my dreams)

- Fiddling with bits (256 x 512)
  - Need about a megabyte of L2 cache
- Doing a memory full of work (64K x 64K)
  - About 5 minutes in a couple of Gigabytes.
- Local disk worth of work (256K x 256K)
  - 64 of these – ideally about 16 cores taking 20 minutes.
- Doing a file-server's worth of work (2M x 2M)
  - 512 of these – ideally about 50 machines taking a few hours.
- So surely 2M x 2M matrices mod 2 in 1 day!

# But

- I do not know a language to code this up in!
- I think I know what to do, though.
- A supercomputer is a device that turns
  - A compute-bound problem into
  - An I/O bound problem
- ☹️

# Big $A += B \times C$

- At the larger levels, chop each of  $A$ ,  $B$  and  $C$  up into  $m$  pieces in both directions.
- Then do  $m^3$  operations  $A_{ik} += B_{ij} \times C_{jk}$  on the smaller pieces.
- Stick the  $m^2$  pieces  $A_{ik}$  together again at the end.

# Gaussian Elimination needs the “Echelize” routine.

Input – one matrix  $M$ .

Output – three matrices –  $Q1$ ,  $Q2$ ,  $R$  – and a list  $P$   
of columns.

Puts  $M$  into echelon form, and outputs  $Q1$  as the  
row operations done –  $Q1.M$  is in echelon form.

$Q2$  is the nullspace found as it goes –  $Q2.M = 0$ .

$P$  is the list of columns where the pivots are

$R$  is the rest of the matrix.

# Small Echelize

- Start with matrix  $R = M$  and  $Q = \text{identity}$ .
- For each  $i$ , do row operations to make the first remaining non-zero column have 1 in the  $i$ 'th row and zeros in all other rows.
- Keep note (in  $P$ ) of the columns used
- When remaining rows all zero, stop and output  $P$ ,  $Q$  and  $R$ .
- Output top and bottom half of  $Q$  in separate files  $Q1$  and  $Q2$ .

# Echelize is recursive

- To echelize a big matrix . . . .
- Chop it into pieces ( $a \times b$ , say)
- Use echelize  $a.b$  times on smaller things
- And also uses multiply and add a lot (so this is performance critical)
- Other administrative routines . . . .
- And then stick the bits together at the end.
- Complex but possible

# Pivot files

- New type of meataxe file – a list of pivots – an  $r \times c$  matrix with each row having a 1 in position  $X_i$  in an otherwise zero row.
- The  $X_i$  must be in increasing order.

# Pivot Extract

- Input and output are format **D**.
- Selects a particular sequence of columns according to the pivot list P.
- I believe this output should be negated.
- Also output the non-pivotal columns.

# Pivot combine

- Combine two (or more) pivot files.
- Most numbers must be renumbered
- The pivot lists are merged.
- Can also merge the rows of matrices in the same way at the same time.
- If properly designed, works with one entry or row at a time and does not need chopping, even in big cases.

# Finding Imprimitivity and Tensor factorization

- Doesn't look at all easy in general
- **If** elements of order fairly large compared to the degree can be found
- We can afford to look through all vectors in an eigenspace and we should be OK
- Imprimitivity is easy once this is given
- Tensor is not so immediate. W.I.P.