10 Years of meataxe development.

Richard Parker.

INI, Cambridge 28.1.2020

Early history.

- Jon Thackray and I started working on computers with matrices in 1977-8.
- Until 1995 computers just got bigger, faster, better, cheaper, but the basic techniques changed very little. Speed depended on doing as little work as possible to achieve the desired result.

1995-2015

- In ~1995, hardware development changed direction. The laws of physics started to obstruct further increase in speed, and later performance improvements depended on doing more things at once.
- Everyone just ignored this issue and hoped it would go away.

Meataxe work split.

- Functionality was added by various people to make it ever easier to compute decomposition numbers.
- Whereas I was *obsessed* by performance, with ease-of-use not even on my radar.
- At some point we need to find a way to reunite the parts.

2015 Onwards

- New specialized devices – GPU, AI accelerators have popped up.
- They have not yet stabilized – will the best be just like this in a few years time? No!
- I looked briefly at GPU's. Moving data about is harder than doing "work".
- I suspect that is a general observation.

But x86 **has** stabilized.

- Perhaps because there is so much legacy software, but also because of the sheer size of the beast, changes are coming more and more slowly.
- Instruction set, cache-sizes, shared memory architecture, vector sizes etc. have changed little in five years, and probably will change little in the next five.

So now . . .

- x86 changes have almost stopped. Tomorrow's will look much like today's.
- Things have perhaps not stopped in a very sensible place! But they've stopped.
- We now have time to look in detail as to what computers we have now, and will probably have over the coming years.

Software today.

- Most talks here this week are about some algorithm and a fairly straightforward implementation. . .
- which will often run roughly five times faster now than it did in 1995.
- A better algorithm can speed things up by an unbounded amount.

Or . . .

- You can work, sweat, toil over your implementation of the same algorithm for years and years and make it run 5,000 times faster on today's x86 than 1995.
- It is rarely worth the effort!.

But I got interested.

- No sane person would spend eight valuable years of their life speeding MeatAxe up by a factor of ~500. OK.
- But did it anyway! Why?
- Because I feel it is a good idea to see what we can do with the computers we actually have.

A glimpse of the future?

- Computer hardware is radically different now compared to 1995.
- But by-and-large programmers continue to ignore this. This, too, is insane.
- My aim today is to give you a glimpse of what the current hardware is **really** like, and how one might set about programming it.
- You're not going to like it!

My Starting point 2011.

- In 2011, Jürgen Müller, Max Neunhöffer and I tried to get a lower bound for the time to multiply random huge matrices mod 2 on the Essen IEM 12-core machine
- We could find no reason to believe a 100-fold speed improvement impossible.

Our estimates

- 12 cores doing 2 128-bit XOR instructions per clock cycle at 2.5 GHz using grease level 4
- 12 x 2 x 128 x 2.5 x 4 = 30,720 elementary operations per nSec.
- Should multiply two square random 470,000 dimensional matrices mod 2 in an hour.
- Wow. Can we actually **do** this? That would be **amazing**.

Well . . . not quite.

- My 2-week trip to Essen in February 2012, working with Reiner Stachewsky, achieved a 350,000 multiply in 80 mins.
- Both before and after this, we have learned a lot from our numerous mistakes.
- Let me teach you some of them.

Two kinds of things to learn.

- Issues related specifically to matrix multiplication and Gaussian Elimination over “small” finite fields.
- General problems using ancient tools to program modern computers, along with suggestions as to what better tools might look like.
- I’ll take them in that order.

Allow for local sparsity.

- In 2006, with P.E.Holmes, we tried what we thought would be a huge improvement on mod 7 arithmetic. We assumed the matrices are random, but in actual use there are zeros all over the place.
- It was slower in practice that the old system!
- The old system was better with those zeros. The new one better with non-zeros. Not good enough!

Allow for format changes.

- Matrix multiplication is cubic (for now).
- It is worth doing as much as possible of the work (quadratically) first to speed up multiplication.
- For C = A x B, there are 4 steps.
- Convert A into “Afmt”,
- B into “Bfmt”.
- Multiply Afmt A by Bfmt B giving Cfmt C
- Obtain the answer from the Cfmt.

Formats

- Depend on the field, allow a diverse set of algorithms to be used for different fields.
- Afmt and Bfmt do such things as sparsity detection, grease coding and padding to avoid special cases, and bit-slice/float format conversions.
- Cfmt also completes the reduction mod p.

Extension Fields

- The lowest levels only need to cope with fields of prime order.
- As far as I know, the fastest way to deal with extension fields is, as Martin Albrecht taught me, to convert each extension field matrix into lots of prime field ones, multiply them pairwise, and assemble the answer at the end.

Example – GF(4).

- We are given a good way to multiply matrices mod 2. We want to multiply matrices AxB over GF(4)
- Every entry is r+sx with $x^2+x+1=0$.
- We make three matrices R consisting of the r values of each element, S similarly s, and T=R+S. For both A and B.

GF(4) continued

- We then compute $R_C = R_A \times R_B$ and similarly for S and T.
- Then make the answer over GF(4) by computing $R_C + T_C$ and $S_C + T_C$ and then assembling them together as GF(4) matrices again.

Where does this idea go?

- We need to be able to extract the coefficients from an extension field matrix, do linear operations on ground-field matrices, multiply ground-field matrices and re-assemble the extension field matrix at the end.
- Result is that we only need high-performance multiplication for prime fields.

Mod 3.

- Mod 2 is XOR. Nothing to add here.
- Mod 3 there is a sequence of six logical operations that adds two coded entries mod 3 (each held in two separate bits). There is a way such that swapping the bits negates the entry, which saves a factor of two on cache over Boothby's encoding.
- Details on request.

Mod 5,7.

- There is a vaguely similar sequence of 13 logical operations that can be used to add mod 5 using data in 3 bits.
- A scheme with 15 operations similarly adds mod 7. This is based on the half-adder well known to hardware engineers.
- I have not implemented either. Details on request.

Add/Subtract codes.

- Given a finite Abelian group A, define an AS-code for A as a subset S of A such that every element of A is of the form $s_i - s_j$ for some s_i , s_j.
- Example. In the cyclic group of order 41,
- { 0 1 2 3 4 10 15 20 } is an AS code.
- Plan is to make these multiples of each vector of B upfront, then do one (vector) add and one subtract to do the work. Used for 5-193.

Floating point to 27,397,079 .

- Steve Linton has recently implemented the high-performance arithmetic using the floating-point hardware.
- 197-1669 using single precision
- Then up to 27,397,079 using double precision. Speeds are roughly comparable.
- For primes 67-193, this out-performs AS on most machines – but not all, and the margin is moderate, so today 67-193 is still AS.

Strassen etc.

- In the real world, a lump of computing hardware takes up cubic centimeters, needing power in, cooling out, data in, data out . . . they all compete.
- Strassen saves “work” but uses more data movement, and in practice, it gains surprising little.
- I sometimes wonder whether there might be a theorem that, in some model of the real world, Strassen is precisely useless.

General issues.

- I claim the whole software world is determinedly ignoring the design of the modern computer.
- The elephant in the room!
- But let’s try and take our first steps.
- What does, or will, this new programming world look like?

Load Balancing.

- My earliest mistake (1991) was to ignore load balancing. If you use multiple machines, you must allow for some parts going faster or slower than you expect.
- You want to keep them all busy all the time
- Otherwise your program will run only as fast as the slowest bit. This is a surprisingly bad mistake to make.

Missing Piece - Scheduler

- Somewhere in any decent system there must be a way to “submit” bits of work, and for workers to “grab” a bit of work when they finish the previous one.
- This needs to be stable, standard, fast and functionally rich.
- No doubt it will eventually emerge!

Memory Hierarchy.

- We cannot ignore the multiple layers of cache memory. Times per 64 bytes
- L1 cache ~32 KB ~0.25 nSec
- L2 cache ~256 KB ~1.0 nSec
- L3 cache ~16 MB ~30 nSec
- RAM ~64 GB ~150 nSec
- (Disk/SSD) yet bigger and slower.

How do you do something?

- You chop it up into a few smaller pieces, then call a routine that does those smaller pieces.
- Matrix multiplication in meataxe64 has about 11 levels of that.
- And it’s not always easy.
- But Matrix multiplication chops naturally.

Multiply Layers of C = A x B

- On disk so it fits in memory (not done!)
- Do slices of A/C to save memory
- Chop in memory to use multiple cores.
- A few (e.g. 3) Strassen steps
- Extension → ground field
- Format conversions.
- Chop so that C fits into L2 cache
- Chop so that (greased) B fits in L1 cache
- Multiple grease blocks to reduce load/store time
- Multiple vector registers to reduce admin time
- Finally do some actual work

The synchronization problem.

- With lots of things going on at once, suppose you want to do something and use the result to do something else.
- It is a very bad mistake to use data that you haven't finished making. Hardware, OS and compilers may reorder things (for performance!) so that "finished making" is not a simple concept. Einstein!
- We again want a solution that is stable, standard, fast and functionally rich.
- No doubt it too will eventually emerge!

Assembler.

- The modern computer has all sorts of wonderful instructions – vectors, conditional moves, for example – that compilers seldom use, partly because the language has to be portable.
- OK, the “assembler aware” programmer can, with great expertise and work, often persuade the compiler to do what is wanted.
- But I find it easiest to just write it myself. I write the bottom layers in assembler.
- Which has not noticeably changed in 40 years!

Catch-22

- Nobody in their right mind writes in assembler because it is so hard.
- Nobody produces a better assembler because nobody in their right mind writes in assembler.
- Surely we've learned **some** things in 40 years that assemblers could use!
- If only assemblers some of them.

My general conclusions.

- The computer industry generally has been slow to react to the change in direction, and software development is now **way** behind the hardware.
- This makes it needlessly hard to write high-performance software.
- There are early signs that this might be (slowly) changing.

Results - Gaussian Elimination.

- Trying to use multiple cores to do Gaussian Elimination using my primitive thread-farm was a huge challenge, but with the assistance of Steve Linton, Gabrielle Nebe, Alice Niemeyer and Jon Thackray, we managed to get it to work in the end.
- “Full” Gaussian (with transformation matrix) is now about the same speed as matrix multiplication.

Results – how fast is it?

- Biggest practical job was to chop a 440,154 dimensional module for 3.F22 over GF(25). Took about 3 days on “lovelace” - a 64-core AMD machine in St. Andrews. No Holt/Rees!
- Monster multiplications take a couple of minutes in 196,882 dimensions mod 2 on 64 cores.
- Even mod 1669, 10,000 sized multiply takes 40 seconds on this laptop.
- Up to 1,000,000 dimensions? Probably feasible.

Doing, however, simple jobs.

- It does not yet do Holt-Rees well. Chopping over GF(25) needed tricks to avoid needing this.
- One basic problem is to compute the characteristic polynomial of a huge matrix (e.g. 440,154 over GF(25)) in a time commensurate with matrix multiplication.
- We think we know how to do this, but issues remain.

Polynomial factorization.

- Another issue that will be needed eventually is to be able to factorize the characteristic polynomial once it is computed. This puts intolerable pressure on the thread farm to gain the concurrency needed, and again a design is sketched, but implementation some way off.

Order of a matrix.

- I still do not know how to find the order of a matrix, even given the characteristic polynomial.
- The factorization of p^d-1 needs an oracle, which is one issue.
- Sorting out the p-part of the order is another.
- I do not really know exactly how to do either of these. Perhaps someone can fill me in.
- “If it’s not cubic complexity, it’s not an algorithm at all”.

Higher level functions.

- For example, experienced meataxers have a program that chops a module up into its irreducibles automatically. We really ought to be able to build up layers upon layers of software, but so far each new layer seems to need a lot of work.
- Yet again, we have ideas but no code.

My over-generalization.

- Simple algorithms can often be speeded up, on a modern computer, by factor of 100-500.
- But only by spending 50 times as long writing them.
- Is it worth it? Probably not.

In the meantime . . .

- Steve Linton is gradually trying to put this into GAP.
- If anyone wishes to **use** any or all of this stuff, they are very welcome.
- Just beware than you might need to put in 50 times as much work as you expect!