Before we start

- Hello everyone!
- There’s lots of text on these slides. This is to fill in the details if you’re reading these slides after the talk, and to assist anyone who has difficulty listening during the talk.
- You don’t need to read the slides as I talk.
- Brown is used to highlight important text.
Faster Rasters For All

Graeme Bell
Norwegian Forest and Landscape Institute
Fig. 1: A faster raster.
What are rasters?

**Geometry/Vector Data** - records the shape of areas with shared values

**Explicit Geometry**

**Raster Data** - arrays of values corresponding to regular areas. This example is low-resolution.

**Derived Geometry**
... incidentally...

- The geometry on the last slide came from a Norwegian map service called “Kilden”, built by my colleagues at Skog og Landskap.

- Check it out. It’s super-nice and built on an open source stack.

- Just type “kilden map” into Google.
Rasters & Me

- In my current project, I’m using rasters as a tool for manipulating diverse or time-varying geometry datasets.

- Note: Raster data from geometry looks slightly different to natural ‘sensed’ raster data in GIS (elevation rasters, landsat).
Why you might find this talk interesting...

• Today I’m going to explain how I get GDAL tools to run **100-200x faster** than the default settings...

• ... weeks to hours, hours to seconds ...

• and produce data files **200x smaller**

• ... using only a single, everyday desktop PC.
Why you might find this talk interesting...

• ... I’ll explain how we can make everyone a GIS expert, by sharing high-performance configuration idioms more effectively...
Why you might find this talk interesting...

• ...and I’ll offer a simple technique (with implementation) to let non-programmers work quickly and safely with complex, high performance map algebra, like this...
“...what the heck is that”

- ss.select([fa.all([A==255,B==255,C==255]),fa.all([fa.all([C>=1,C<=254])]),fa.all([B==21]),fa.all([A==21]),fa.all([fa.all([B>=1,B<=20])]),fa.all([fa.all([B>=22,B<=254])]),fa.all([C==0]),fa.all([fa.all([A>=22,A<=23]),B==21,C==23]),fa.all([fa.all([A>=30,A<=60]),B==21,C==91]),fa.all([fa.all([A>=30,A<=60]),fa.any([fa.all([B>=1,B<=20])]),fa.all([B>=22,B<=254])])])
Wait, that’s ‘obvious’...

• Some of the ideas I’m going to mention today are well known to GIS programmers.

• e.g. Splitting big problems into small problems with tiling? Not rocket science.

• The focus isn’t on the detail of these ideas.

• The focus is: how can we flatten the learning curve for high performance raster GIS - ‘faster rasters for all’?
My motivation is dirt.

- My current project combines new & old national Norwegian land maps, to produce an estimate of current land & soil status.
- The output is a transformation using multiple different geometry sources, from different points in time.
- It’s Kinda-Big Data - each geometry table for a point in time is ~ 10-20GB in size.
The first big problem

Here are two representations of the same farm, either in different datasets or at different times.

The shape, size, number and identity of objects in a geometry map may change over time, or vary between related datasets.
Reasoning between datasets and times

• I needed some fixed ‘objects’ to talk about, so that I can reason across datasets (or versions of the same dataset).

• To solve this, I’m using a simple grid model, i.e. a rasterization of the sources using a fixed reference grid and coordinate system. My invariant objects are grid cells.
A raster workflow

• Reproject the source data (if necessary) to a standard coordinate system.

• Rasterize sources into a grid system.

• Transform data (with GIS raster algebra).

• Generate pretty output (DBs, web stuff)

• Grid-based reasoning simplifies things immensely, but created some new problems.
Problem A: How do I make it fast?

- GDAL, the main open source raster toolkit, does not seem to work well with this kind of problem, by default.

- `gdal_rasterize` with a 15GB SQL table of Norway at 10mx10m resolution takes a week with default settings, even on a modern Xeon processor (2013) and with spatial indices on the DB.
Problem A: How do I make it fast?

- A week isn’t so bad - but we’d like e.g. 5x5 or 2x2m resolutions available (25x more work)
- We have multiple datasets to rasterize (another 4-10x more work)
- And we have more processing after the rasterization itself (perhaps 2x more work)
- 500 weeks (10 years) to version 0.1 is not ideal.
- See you all at FOSS4G 2023! :-(
Problem B: How do others use this?

• Knowledge about improving performance in GDAL (and other open source projects) is scattered around the web, and idiomatic.

• If a forum post tells me a weird trick to make the GIS workflow run faster, what happens if I’m not able to maintain it later?

• How can others take advantage of what I learn, for this and other projects?
Problem C:

How can non-programmers build GIS transformations?

• Many experts in farming, land, soil and forests are not GIS programmers.

• How can I make it easy for non-experts to build, verify and maintain big, complex GIS transformations?

• How can I help them make it fast?
Problem A: How do I make it fast?
The project environment

• Simple quad-core desktop system.
• PG9.2, PostGIS 2.1, GDAL 1.10, numpy 1.7
• Connect to PostGIS databases containing new & old geometry tables.
• Transform the tables.
• Store results back into postgres and/or keeping it locally as shp or GeoTIFF raster.
Observation 1: GDAL defaults are slow?

- At the start of my project, I was new to GIS, so I read all I could about GDAL performance settings in blogs, articles, forum comments, and made my own measurements.

- But, I noticed that rasters burned from geometry are unlike rasters from remote sensing. Many idioms and ‘blog-facts’ don’t hold true any more.

- It’s really common to see repeating values, integer values, perhaps only a few distinct data values.
Example: output datatypes

- For example: the default for gdal_rasterize’s output is the Float64 datatype.

- How does that affect the run time and filesize for users that don’t fine-tune their code?

- I ran gdal_translate to re-encode a sample TIF. No compression was used for the output.
Example: output datatypes

- **float64**: 1.44 seconds, 391 MB.
- **byte**: 0.18 seconds, 49 MB. (‘obviously!’)
- **float64** is the default perhaps because it is the ‘most compatible’ type, with least chance of losing precision accidentally ...
- **BUT** normal users may be completely unaware that they can improve performance and filesize by up to 8x for many rasters with just one simple step.
Example: output datatypes

• (Developers: it would be nice if GDAL tried to detect the fastest safe type, rather than always using the slowest...)

Observation 2:
compressed raster $\approx$ geometry

- Geometry-based maps store the boundaries around areas that have a shared set of values. Geometry boundaries record where data values suddenly vary.

- Compression systems record entropy (variation). e.g. Run-length raster compression, a common approach, effectively records where data values suddenly vary.

- Fast lossless compression seems like a great match for rasterized geometry!
**Experiment: gdal_translate**

a test area in Norway

<table>
<thead>
<tr>
<th>Compression</th>
<th>Filesize (MB)</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>48.9</td>
<td>0.18</td>
</tr>
<tr>
<td>LZW, 1, 1</td>
<td>2.8</td>
<td>0.39</td>
</tr>
<tr>
<td>LZW, 1, 2</td>
<td>2.8</td>
<td>0.43</td>
</tr>
<tr>
<td>PACKBITS</td>
<td>2.7</td>
<td>0.17</td>
</tr>
<tr>
<td>DEFLATE, 1, 1</td>
<td>2.4</td>
<td>0.31</td>
</tr>
<tr>
<td>DEFLATE, 1, 2</td>
<td>2.8</td>
<td>0.41</td>
</tr>
<tr>
<td>DEFLATE, 9, 1</td>
<td>1.8</td>
<td>2.64</td>
</tr>
<tr>
<td>DEFLATE, 9, 2</td>
<td>2.1</td>
<td>4.91</td>
</tr>
</tbody>
</table>

fast, small!

Geek note: predictors are BAD for rasterised geometry!
Conclusion

Using fast lossless compression with rasterised geometry is a really REALLY good idea.

Almost no time cost, massive impact on filesize.

Current practice:

The GDAL GeoTIFF raster driver defaults to no compression.

(Developers: it would be great if gdal_rasterize gave a hint to the driver: default to lossless compression?)
Observation 3: gdal_rasterize is not a spatial DB!

- gdal_rasterize runs disproportionately slowly with
  - huge amounts of geometry
  - huge / high resolution raster output
Solution: PostGIS is a spatial DB

- Keep your geometry in PostGIS, render tiles, merge.
- Use **bounding boxes** and **spatial indices** to quickly get the minimum geometry needed to rasterize each tile.
  - Small amounts of raster data = good use of cache, less problems with big-O algorithm performance.
  - Small amounts of polygon data = lower demands on CPU/memory, faster time-to-start.
Observation 4: GIS tools don’t always parallelize well

- Introducing Gnu Parallel, the best UNIX tool you’ve never heard of.
- It allows easy, staged parallelization of any scripted task, with nice default settings that automatically scale to suit your system.
- Great for parallelizing my raster tile builds.
- (and easier than rolling my own solution)
Putting it all together (I)

• Test patch, 10mx10m, ~ 0.5-1% of norway

• default settings of gdal_rasterize:
  • 1103 seconds

• with the techniques I suggest:
  • 5 seconds

  22000% faster
Putting it all together (2)

- All Norway map, 50mx50m
- default settings of gdal_rasterize:
  - 4 hours
- with the techniques I suggest
  - 116 seconds

12400% faster
# Breakdown of effects

<table>
<thead>
<tr>
<th>Approach used for all-norway 50x50 map, desktop 4-core chip</th>
<th>Render time (seconds)</th>
<th>Filesize (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>default GDAL parameters</td>
<td>14212</td>
<td>5400</td>
</tr>
<tr>
<td>with byte datatype</td>
<td>1873</td>
<td>688</td>
</tr>
<tr>
<td>with byte + light compression</td>
<td>1888</td>
<td>47</td>
</tr>
<tr>
<td>with byte + comp + 200MB GDAL_MAXCACHE</td>
<td>1875</td>
<td>47</td>
</tr>
<tr>
<td>with byte + comp + cache+ merged bounding boxed tiles</td>
<td>488</td>
<td>23 (high compression merge)</td>
</tr>
<tr>
<td><strong>RBUILD</strong> (all above + parallelism)</td>
<td>116</td>
<td>23 (high compression merge)</td>
</tr>
</tbody>
</table>
Problem B: How do others use this?
Observation 5: Programs aren’t consistent

• Some GDAL tools use: -co for options, others, --co.

• ... differently named nodata switches ...

• even the program names vary in style: gdalinfo / gdal_rasterize / gdal_polygonize.py

• -q, -v, -d: what console output do you expect?

• This is a huge pain for experts & non-experts alike!

• Especially when subtle typos result in complete failure with no guidance about what was wrong.
Observation 5: Programs aren’t consistent

- Even if GDAL was totally consistent, you will always have variation against other O/S software projects.
- Maybe we should try to hide or simplify this stuff by offering easy groups of defaults.
- McDonalds has this one solved:
  - “SUPERSIZE ME”
- Works on anything: burgers, drinks, fries, ice-cream, entire meals, entire orders, ...
Observation 6: Parallelization can be tricky

- For a 10mx10m all norway map, using ~100 parallel tiles works best for me.

- More tiles? Slight slowdown due to extra overhead.

- Less tiles? GDAL tools slow down disproportionately.

- We can probably guess an optimum for a map...

- How do you make that kind of expert information accessible to all users - or hide the problem from them?
Observation 7a:

performance tricks are great

- compression, parallelisation, datatype settings...
- gdalwarp or gdal_merge.py? (gdal_merge!)
- gdal_polygonize.py or gdal_trace_outline? (depends...)
- DB hacks:
  - Unix sockets instead of TCP/IP --> 30% speed boost!
  - ‘tables’ flag in the PG: parameter, avoids costly DB scans.
- Bug workarounds!

(e.g. GDAL geotiff driver generates wrong nodata values - trac #5115)
USE

*ALL* THE PERFORMANCE TRICKS!

imgilp.com
Observation 7b: performance tricks are a PAIN!

- Much of what is posted on the internet is wrong or out of date, or is specific to particular use cases...
- And how do you learn about new tricks?
- How do you know when old tricks are outdated?
- How do you maintain old code which may have had tricks used in it... or not?
- Maybe we should be separating out / abstracting our tricks and idioms from the rest of our scripts?
Observation 8
reproducability, metadata

- It’s easy to make rasters with GDAL.
- But it’s hard to figure out where the heck “map5_VERY_IMPORTANT_DONT_DELETE.tif” came from.
- Life would be much easier if every raster (and geometry set, for that matter) came with the recipe and links to ingredients for baking a fresh copy again.
- Especially if you’re a scientist or are working with important data.
So... I wrote a little program to try to solve all these problems
A solution: rbuild

- It provides a controllable optimised workflow.
- parallelization, compression, optimisation tricks cleaning up between runs, rasterizing source geometry, performing calculations, merging, building overviews, re-polygonizing to shapefiles, defining metadata - parameters, build time, ... (JSON) adding data/metadata to PostGIS / PG Raster...
- It hides all the quirks and problems of each stage.
Workflow, build files

- Rbuild’s behaviour can be controlled by ‘build files’ that contain groups of re-usable, useful settings.

- e.g. ‘performance_fastest.bf’, ‘actions_do_everything.bf’.

- Instead of passing on tricks or code idioms on forums, you just create a buildfile and share it.

- Users don’t need to check if their performance tricks for gdal are becoming outdated and unhelpful; they can just keep their buildfiles up to date instead.
A named system of buildfiles removes repetition and hackiness from writing GIS scripts, and helps with maintainance and sharing knowledge.

You can choose to put only the new aspects of your project in a buildfile.

You could re-use buildfiles from existing projects that define data sources, output locations, GIS calculations, performance, behaviour...

It’s like using programming libraries, instead of rolling your own code each time.
moving on from rbuid...
Problem C: How can non-programmers build GIS transformations?
This is what would happen if I had to think up soil descriptions myself:

- `int brown_ish_ness?`
- `boolean is_full_of_worms?`

I definitely need an expert for this.
Step 1: Keep It Simple

- If you’re going to ask non-programmers to help you transform maps, it’s good to remove from consideration things like:
  - how to combine areas of varying shape
  - how to combine areas of varying size

- Tip 1: Regular grids are simple and great!
Step 2: Let them make a spreadsheet!

- Using an excel spreadsheet file as the source code for a two-stage compiler is definitely the weirdest idea I’ve ever had.

- But everyone knows spreadsheets, right?

- I get the transformation defined like this:

<table>
<thead>
<tr>
<th>rule number</th>
<th>in_map1</th>
<th>in_map2</th>
<th>out_type</th>
<th>out_confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1..5, 8..10</td>
<td>*</td>
<td>“soil”</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td>1..99, 101</td>
<td>“not soil”</td>
<td>50</td>
</tr>
</tbody>
</table>
Compile a spreadsheet!

- I compile the inputs into high-speed **numpy**.
- That way, the expert doesn’t need to design fast GIS code. My compiler does that.
- New GIS trick? Recompile the transform.
- New GIS platform? Recompile the transform.
- I convert outputs into SQL update statements.
- I implemented this using **pyparsing** in Python.
- It seems to work really well!
AND NOW

HERE ARE THE TOYS
Open Source outputs from this project

- **Rbuild**, the workflow system I described.
- Available in 7-14 days time* at [http://github.com/gbb](http://github.com/gbb)
- * big project deadline next week, sorry everyone
Open Source outputs from this project

- **Ruleparser**, the spreadsheet system.
- Available in 7-14 days time at [http://github.com/gbb](http://github.com/gbb)
Open Source outputs from this project

- An improved gdal_calc.py.

- It allows more flexible nodata handling.

- It runs 15-30% faster than the normal version.

- Available right now at http://github.com/gbb and submitted to GDAL as trac #5130.
A bug-fixed and optimised `numpy.select()` for use with `gdal_calc`.

- It runs 2x faster generally and 5x faster on problems like typical GIS rule-based raster transformations.

- Available now at http://github.com/gbb, pull-requested for numpy master, and built into rbuild.
Open Source outputs from this project

- An optimised version of `numpy.any/all()` for `gdal_calc`.
- When used with `gdal_calc`, it runs **30x faster** than the version in the current stable `numpy 1.7`.

- Available now at [http://github.com/gbb](http://github.com/gbb) and built into `rbuild`. 
Open Source outputs from this project

• Various bug reports & performance suggestions to open source projects.
Thanks to

- My institute, for letting me share this as open source.
- All the open source developers in GIS, postgres, python, and Gnu/Linux.
- Most of my work is just re-using your work.
- You guys are awesome. Thanks for sharing!
Thanks for listening!

slides : graemebell.net
code  : http://github.com/gbb
Bonus slide: Crikey, another build system?

- Well... I don’t know about you, but I certainly needed it... :-)

- It would have been possible to build the system around ‘make’ instead of ‘bash’, but then you get problems with significant whitespace that make it hard for non-programmers to play with the scripts and improve them.

- ‘Ant’ would require knowledge of java and XML to configure and extend, rather than simple plain text, so it is really best suited to building java projects.

- Neither approach would offer such a user-friendly method for adjusting workflow behaviour at run time -A yes, -B no.
Images used in talk

- Fast car:
  - http://2.bp.blogspot.com/-njPmdU0lZ4U/UHOKOyRKNCI/AAAAAAAAENI/p-HRMi4arng/s640/fast+car+5.jpg

- Smiley:
  - http://cse1.net/recaps/img/12-smiley-large.png

- “Memes” - generated using imgflip.com, original sources unknown.
Random interesting things

- Those results are not necessarily representative for the case of rasterized-geometry rasters.
- Numpy, pyparsing, gdal_calc, bash, Gnu Parallel.
- Kilden maps! :-}