R Users Hate Him!

Learn his Mind-Blowing Hacks to Speed Up your Simulations!

Isaac Ray

2024-04-10

Bag of Tricks

- No Language Required
  - Benchmarking
  - Containerization
  - Pay to Win
- R exclusives
  - ‘Common’ advice
  - Byte Compilation
  - BLAS / LAPACK
  - Explicit parallelism in R
- Low Level Languages
  - Maximizing Rcpp performance
  - Compiler Directives (OpenMP, OpenACC)
  - CUDA / GPU support
  - Advanced topics

POV: The average STAT graduate student

Figure 1: Your advisor has asked you to run your method on a grid of a million hyperparameters, but it takes 10 minutes to run once

Motivation

- Most of us research/develop/apply our statistical techniques in a high level scripting language (R, Python, etc.)
- When it comes time for simulations and real data analysis, your implementation is painfully slow
- **What is the path of least resistance to make things go faster?**
  - 'Least resistance' $\neq$ 'easy and straightforward'
- Straightforward (but challenging) example: Gaussian Process Predictions

Math on a slide!? Avert your eyes!

For simplicity we will consider a dense/full/exact Gaussian Process

- p-dim covariates $X \sim \mathrm{Unif}(0,1)$
- n observations Y, X where $Y \sim \mathcal{N}(0, K(x,x)) + \mathcal{N}(0, \tau^2)$
    - We will use a boring squared exponential kernel
- We will assume we know other parameters like length scale (and they're fixed)
- Unless otherwise specified, $n, n^* = 2^8 = 256$

Naive implementation

- `simulate_GP`: draw X, X^* from unif, calculate $K([X, X^*], [X, X^*])$, draw Y, Y^* from `MASS::mvrnorm`
- `predict_GP`: Calculate $K(X^*, X)$, $K(X, X)^{-1}$, do multiple matrix multiplications to get Kriging predicted mean for Y^* and covariance matrix
- **Intuition**: which do you *believe* will take the longest?

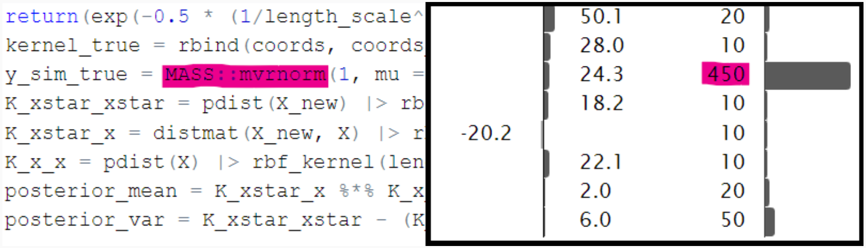
Part 1: Language Agnostic Tips

Benchmarking: Look before you Leap

"Premature optimization is the root of all evil."
- Donald Knuth

- You have to *know* **what** is slow before you can make it faster
- We are scientists! Take advantage of your skills
  - The timing of a model run behaves like a random variable
  - You wouldn't trust a claim using a single observation
- Benchmarking is easier than ever!
  - `microbenchmark`, `system.time()`, `tictoc`, `Profvis`, etc
  - `Timer` class in `Rcpp`, `RcppClock`, `rcppgeiger`
- Don't blindly trust Big-O notation; constants add up quickly

Naive Implementation: What's slow?



```
return(exp(-0.5 * (1/length_scale^                     50.1     20
kernel_true = rbind(coords, coords                     28.0     10
y_sim_true = MASS::mvrnorm(1, mu =                     24.3    450
K_xstar_xstar = pdist(X_new) |> rb                     18.2     10
K_xstar_x = distmat(X_new, X) |> r     -20.2                    10
K_x_x = pdist(X) |> rbf_kernel(len                     22.1     10
posterior_mean = K_xstar_x %*% K_x                      2.0     20
posterior_var = K_xstar_xstar - (K                      6.0     50
```

Figure 2: `profvis({native_impl(N_benchmark, N_star_benchmark)})` ; left column is memory changes, right column is approximate timing

Custom MVRNORM: big difference!

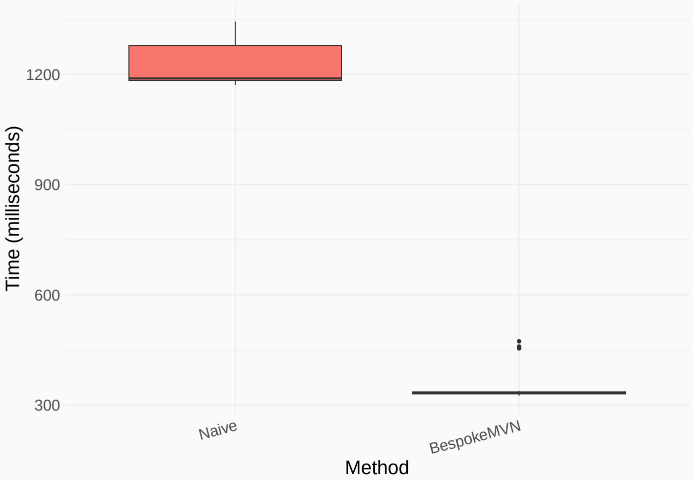


Figure 3: Swapping MASS::mvrnorm for a simple function in R

Containerization: A Whale of a time

- We must ensure our experiments don't have confounders
- By **containerizing** our code we can ensure reproducibility down to system libraries
- Added bonuses: *portable* and often using *optimized libraries*!
- Most well known software for this is Docker
  - Due to privilege issues, often unavailable on HPC machines like HPRC or Arseven
- Solution: **Singularity** (or CharlieCloud, Podman, etc.)

Pay to Win: Sometimes bigger is better

- **Optimized** code STILL maxing out your resources? Tired of creative problem solving?
  - Time for more compute! (AKA: throw money at the problem)
  - Note “optimized”: slow / single threaded code will not benefit
- If you are running things locally, move to a cluster (HPRC, Arseven, ACCESS, etc.)
- Understand the nature of your bottleneck $\rightarrow$ determine what hardware upgrade(s) will improve performance

Part 1.5: What's a computer? (Not an iPad Pro)

High Level Hardware - Central Processing Unit

- **CPU**: ‘Brain’ of your computer, *coordinates* the execution of your program (and does most of the heavy lifting)
    - Examples: ‘Intel i9-12900HK’, ‘AMD EPYC 7763’, ‘Apple ARM M1 Pro’
- Important parameters: **clock speed**, **core count**, cache sizes, memory bandwidth, etc.
    - Desktop CPU vs. HPC CPU $\iff$ Race Car vs. School Bus
- Most statistical applications can be parallelized at some level

High Level Hardware - Random Access Memory

- **RAM**: Where your CPU puts things it might need “soon-ish”[1]
- Important parameters: **size** (GB), frequency, latency
  - Unless you’re deep in the weeds, more GB == better
- Memory locality: the sooner you need it, the closer to the CPU it should be

[1]RAM is the compromise between cache and disk; data your CPU is not actively working on but could need at a moment’s notice

High Level Hardware - Disk

- **Disk**: Where your CPU puts things for long term storage
- Important parameters: **size** (GB/TB), throughput, latency
  - If you don't have enough, you can't store your stuff!
- NVMe SSD $>$ SATA SSD $>>$ HDD $>>>>>$ Network[2]
- Minimize Disk I/O; work with things in RAM (databases, lazy eval, batching)

[2] Cloud/off site; local NAS can be high throughput

CPU & RAM performance on the human scale

Table 1: Assuming a midrange 2020 CPU on one thread

Task	Approx CPU time	Human Equivalent
Add 2 integers	1e-10 seconds	1 second (generous)
Multiply 2 integers	1e-9 seconds	10 seconds
Divide 2 integers	1e-8 seconds	1.5 minutes
Access L1, L2, L3 cache	1e-9, 1e-8, 5e-8	Desk, Drawer, Fridge
Access RAM	1e-8 seconds	Down the street
Access NVMe SSD	1e-4 seconds	A week
Access HDD	1e-2 seconds	A year

High Level Hardware - Graphics Processing Unit

- **GPU**: the ultimate ‘Single Instruction, Multiple Data’ hardware
- Important parameters: **Core count**, **VRAM**, tensor cores, instruction set, memory bandwidth, clock speed, link speed, ... too many to list; hard to compare!
- Only makes sense for *big* problems; heavy overhead
- Rare case where oversubscription is *good*
  - Most often, bottleneck will be transfer: throw as much work at it as possible

Hardware, Bottlenecks and You

All of the following assume your implementation is *already optimized*

- CPU bottlenecks: FLOPS (desired), ‘cache trash’, branch miss
    - For the latter 2, technical code restructuring is needed
- RAM bottlenecks: OOM (most common in R), bandwidth saturation
    - OOM destroys performance; add RAM or divide & conquer
- Disk bottlenecks: IOPS, throughput, latency
    - Soda fountain: cashier speed, flow rate, time to dispense
- GPU bottlenecks: FLOPS (never seen it), VRAM, interconnect, **host transfers**
    - Excessive syncs to host destroys performance

Part 2: 'R'evving up performance

“bUt haVe YOu tRiED veCToRiZIng iT”?

There is lots of folk wisdom about what makes R fast/slow; these are true

- ‘Vectorized’ functions are faster than ‘for’ loops (if it is calling C code!)
  - ‘for’ loops are slow because **function calls** are slow; little gain from `apply`
- You should *allocate space* for objects all at once rather than append
  - Similarly, avoid repeated copies; `c`, `append`, `cbind`, `rbind`, `paste`
- If you’re creating temporary objects, `rm` and `gc` to avoid OOM
- Data structures matter; use faster ones if you can get away with it
  - Atomic > Vector > Matrix/Array > List > Data Frame
  - Better alternatives: `tibble`, `data.table`, `Matrix`

Swapping out data structures: did it matter?

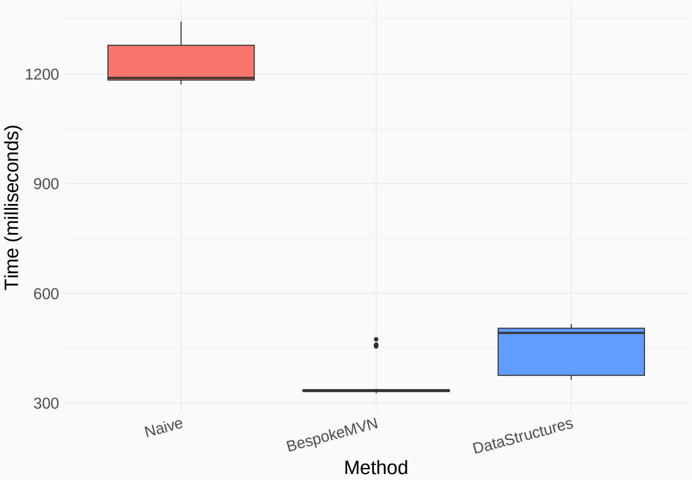


Figure 4: Swapping `matrix` for `Matrix`, swapping in `tcrossprod`

Byte off a chunk of time

- The base R `compiler` package is used on almost all functions in R **packages**
- Unlikely to make a big difference if your code is already vectorized
    - Can also `enableJIT(3)` for minor gains
- More important for functions with `for` loops
- Always worth trying because it's basically free:

```
new_func = compiler::cmpfun(orig_func,
list(optimize=3))
```

Our code is vectorized, no biggie

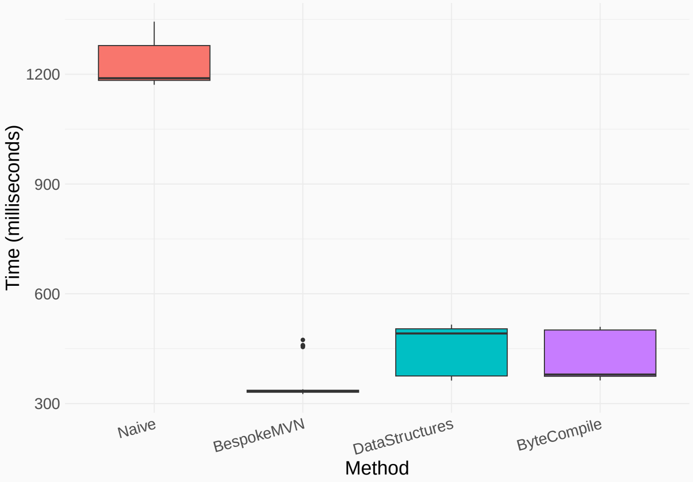


Figure 5: Byte compiling the ‘DataStructures’ implementation

They BLAS on my LAPACK til I Netlib

- Matrix functions call a **L**inear **A**lgebra **PACK**age which calls **B**asic **L**inear **A**lgebra **S**ubroutines
- Base R ships with ‘Netlib’ implementations of these
  - Portable, stable, free to distribute
  - Horribly, terribly, snail’s pace slow
- ***Good news!***You can just use a faster one! But be warned...
- Many choices: MKL, OpenBLAS, ATLAS, vecLib, GotoBLAS, *MAGMA*, *NVBLAS*, and more...
  - My rec: OpenBLAS on Linux, vecLib on M1/2
  - Most of these are implicitly parallelized; careful of oversubscription

Behold, the Power of BLAS/LAPACK swaps!

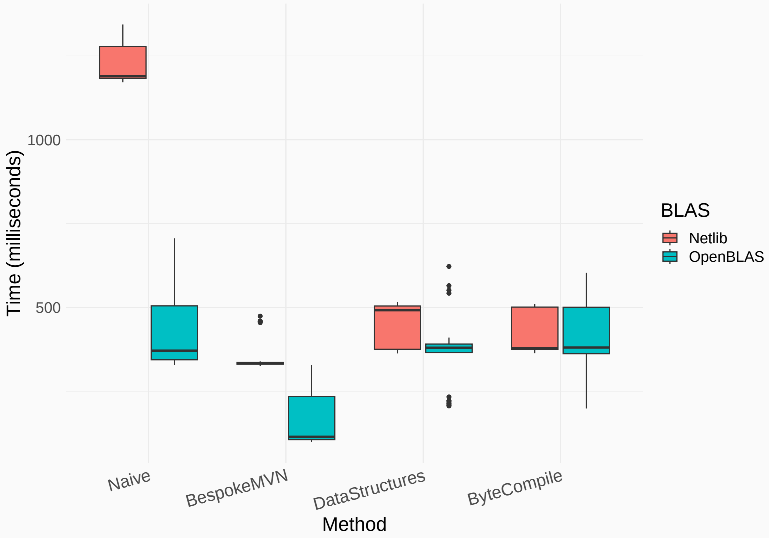


Figure 6: Experiments run on the Grace cluster with 16 cores

Parental Advisory: Explicit Parallelism

- Sometimes the slow part of your code is not related to matrices (coarse grain)
- 'Explicit' parallelism: you tell R how to break up your problem
    - Unless your problem is **embarassingly parallel**, this could be nontrivial!
- Very expensive *memory* overhead, easy to get OOM
- Ecosystem is quite mature now: `parallel` (clusters, forking), `furrr` (map/reduce), `Rmpi` (multinode, broadcast/scatter)
- Easiest option: `%dopar%` and `%doRNG%` with `foreach`; not highest performance

otheR Considerations

- Default file input is slow; I recommend `rio` for the majority of formats
  - csv, xls, Google Sheets, xml, json, Matlab (.mat), Stata (.dta), SAS (.xpt), SPSS (.sav), etc.
  - If you're saving / loading data files, binarize using `saveRDS` and `loadRDS`
- If your data is >10GB, it might be time to stick it in a database
- Some linalg functions are faster (up to constants) (`crossprod`, `tcrossprod`, `backsolve`, `chol2inv`, etc.)

Part 3: Drop ‘it’ (language) low

Unofficial Dirk Eddelbuettel Fan Page

- Everyone probably knows about `Rcpp`, but can we make it faster?
  - Requires actually writing C++ code... easy to shoot your foot!
- Compiler flags: `-Ofast`, `-march=native`, `-Mlarge_arrays` for big data/bioinformatics
- I recommend `RcppArmadillo` over `RcppEigen` because the latter **will not benefit from BLAS/LAPACK** swaps!
- Hard to give specific advice because C++ implementation is complicated and varied, feel free to come chat with me about it

Compilers: the magic of a black box

- A major benefit of writing in C/C++ is the brilliance of compilers
- **Compiler Directives** give you a portable[3] way to write more performant code
- 2 big camps: OpenMP(more popular) and OpenACC (better GPU support)
- Big idea: add simple flags like `#pragma omp parallel for` before a loop and get parallelism 'for free'
  - Nothing in life is free, especially not compute performance. There are graduate courses in CS just on using these tools
- Other approaches: teams (recursive dispatch/greedy searches), SIMD (AVX on CPUs)

[3]except that Apple doesn't like OpenMP, Windows doesn't like pthreads, who knows what ARM is doing, etc....

How to GP-Use a GPU

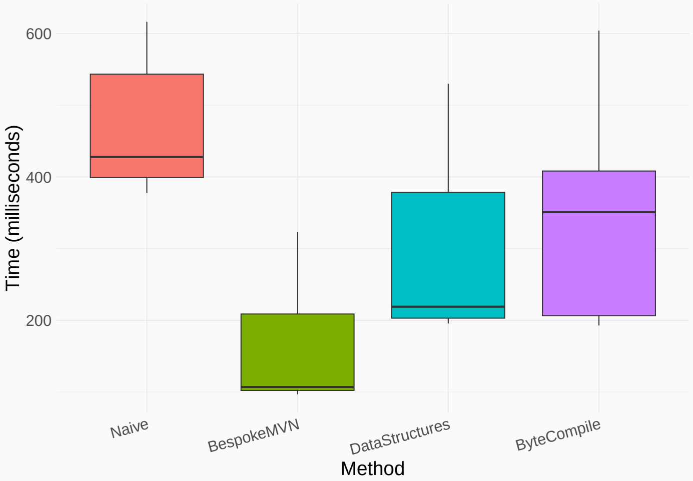


Figure 7: Experiments run on a (totally unecessary) A100 GPU Grace node using NVBLAS drop-in

Cool but Complicated Things I Don't Have Time For

- Multi-Node computation (use MPI)
- Explicit Multi-GPU computation (implicit, just add `NVBLAS_GPU_LIST ALL`)
- PRNG streams (a.k.a. the dangers of the deep state)
    - Big picture: seed splitting. Check out JAX's approach
- Hand rolled CUDA code aka kernels (would take an entire workshop)
    - Big picture: take existing CUDA library, expose first by `nvcc` to a shared object, `dyn.load()` the result and call with `.C`

Wrapping Up

- You can get **major improvements** with **minimal/no** code changes by having a better environment
  - For larger n, I saw up to 476x improvement over naive on GPU
- Benchmark and profile things religiously: things which are slow may surprise you
- Make your software fast so that people will want to use it

Questions?

Appendix: CUDA compile & bind

```
nvcc -O3 -arch=sm_35 -G -I${cuda_includes_path} \
    -I${R_includes_path} \
    -L${R_lib_path} -lR \
    -L${cuda_lib_path} -l${cuda_library_used} \
    --shared -Xcompiler -fPIC -o ${output_file}.so ${input_
```

Appendix: Launching with NVPROF & OpenBLAS

On Linux:

```
ARCH=$(uname -m)
update-alternatives --set "libblas.so.3-${ARCH}-linux-gnu"
update-alternatives --set "liblapack.so.3-${ARCH}-linux-gnu

LD_PRELOAD=${libnvblas_so_path} R CMD BATCH script.r
```