Math 532 Final Project

Antony Sikorski

Table of Contents

Introduction

Everything started midway through the semester, when my high performance gaming computer was outperformed by Dr. Nychka's MacBook, which had a plethora of tabs and programs open. For reference, I have a 2021 ROG Zephyrus M16 Laptop, with a i9-11900H processor and an Nvidia 3060 graphics card. Dr. Nychka has a 2020 MacBook Pro with an Apple M1 Chip that unifies both the GPU and the CPU, and it performed a Cholesky decomposition roughly 10 times faster than my computer did. Since his computer should have been slower in theory, I was naturally curious, and learned that he had optimized his linear algebra libraries, and effectively sped up R on his machine by replacing the default BLAS library with the free OpenBLAS 3.19 library . Some discussion led to me investigating how to best speed up matrix operations on my device as well.

Speeding up linear algebra operations is an incredibly useful concept for those working in statistics, data science, machine learning, and a number of other fields. Statistical model frequently rely on large matrix decompositions and other operations, which are often some of the most computationally expensive steps. The `fields` package is no exception, with functions such as `spatialProcess` and `mKrig` being heavily reliant on the Cholesky decomposition of a matrix. Initially, I looked into the `fieldsMAGMA` package, an upgraded version of `fields` that allows for communication with the MAGMA library in order to outsource linear algebra operations to the GPU of the computer. Unfortunately, I was not able to install this package on my Windows system due to there only being instructions for MacOS and Linux systems. Regardless, I will discuss the usefulness of this package in this report. (Those who are only interested in the speedup should skip directly to "An Easy Speedup for Windows Users")

As an alternative method, I found an extremely simple way to speed up the linear algebra operations in R by making use of the `MKL` package developed by Microsoft. This method requires absolutely minimal computer knowledge, and will be very simple for someone who is comfortable with RStudio. The `MKL` package allows one to "parallelize" matrix algebra on their CPU by utilizing multiple threads and cores of the CPU, rather than using only one thread, as is default for the latest version of R. I provide instructions for the implementation of this speedup, along with a performance comparison via some code that times a number of important matrix operations before and after the upgrade. To conclude the report, I discuss some future work that I will be conducting in this area next semester. All of the code used for the timing comparisons is provided in the Appendix of the report.

Discussion of fieldsMAGMA

Overview

The package `fieldsMAGMA` is essentially a speedup to the `fields` package, because it allows for communication with the `MAGMA` library. `MAGMA` stands for Matrix Algebra on GPU and Multicore Architectures, and the library uses the heterogenous structure of modern computers (uses both the CPU and GPU), to quicken linear algebra operations. Many functions in `fields` use the Cholesky decomposition within them to calculate likelihood, which happens to be the most computationally expensive step. `fieldsMAGMA` allows these functions to be performed much quicker, while obtaining the same results that one would if they were just using the `fields` package. The package requires one to have already downloaded the `MAGMA` library, and to have a graphics card that has a `CUDA` compute ability of 2.0 or higher. For those who do not know, `CUDA` is a parallel computing platform that lets software on your computer use the GPU for certain operations. After installing these two, the only installation instructions provided for `fieldsMAGMA` are for MacOS and Linux systems, and they are quite complex. I did not install the package, but I provide the timing results from the tech report that was released along with the package.

The full report can be found here: https://opensky.ucar.edu/islandora/object/technotes:532

MAGMA can be downloaded here: https://icl.utk.edu/magma/software/index.html

CUDA can be downloaded here: https://developer.nvidia.com/cuda-toolkit

Timing Results

The timing tests in the tech report were performed on a 2014 MacBook Pro with an Intel i7 CPU and a NVIDIA GeForce GT 750M GPU. I assume that the timing results were done more rigorously than mine (I was working on homework and had many programs open when mine were done), although the computer is by default slower than mine. The presence of these factors means looking at the raw difference in time will not be very useful when comparing my method against `fieldsMAGMA`. It is much more important to focus on the factor by which certain operations are set up.

The timing and speedup results were provided for the Cholesky decomposition of a well behaved covariance matrix, the `mKrig` function, and a "sample spatial workflow". All of these were done using the `CO2` data set that comes with the `fields` package by default, and the amount of data points was varied from 200 to 10,000 observations (resulting in covariance matrices with up to 100,000,000 entries). The `mKrig` function uses the Kriging algorithm, which is of computational complexity $O(n^3)$, with the toughest step being the Cholesky decomposition. The tech report was published prior to the writing of the `spatialProcess` function in `fields`, so the spatial workflow is essentially quite similar to this function; it simply lacks a few additional steps that are included for convenience in `spatialProcess`. The spatial workflow consists of estimating λ, the smoothing parameter, performing Kriging with λ and a fixed θ (scale parameter of the exponential covariance model), calculating a prediction surface, and finally estimating the standard error of the surface. Although the number of Cholesky decompositions required changes for each maximum likelihood optimization (same with the `spatialProcess` function), in the case of the report, 11 were necessary. The performance graphs for varying numbers of observations (Figures 1-3) can be viewed in the report, but I will provide a summary table below for the timing results for 10,000 observations:

Table 1: Performance improvements for functions using 10,000 data points (fieldsMAGMA speedup)

Function	Default Time (s)	Speedup Time (double precision) (s)	Speedup Factor (double precision)	Speedup Time (single precision) (s)	Speedup Factor (single precision)
Cholesky decomposition (chol)	126.5	12.9	9.84	1.8	69
Kriging (mKrig)	130	17.8	7.3	7	19
Spatial Workflow	1530	294	5.2	179	8.57

The timing for the largest number of observations is most important since we aim to approach theoretical computational limits. It is also important to note that the sped up results here that are comparable to mine are the values from the tech report where the calculations were done with machine double precision. In the report, there is option for using single precision, which speeds up the process even more, but my results are done with double precision by default. Regardless, we see incredibly impressive speedup values, especially considering that this is sent to a low end graphics card. Naturally, the more complex functions that have more steps other that do not include matrix operations have lower speedup factor values. A more powerful and modern graphics card would most likely achieve speedups of 100x or more. Later in this report, we compare these results to my own, CPU based method.

An Easy Speedup for Windows Users

Overview

Unfortunately, my lack of computer savvy-ness did not allow me to translate the instructions for how to install `fieldsMAGMA` into something that Windows users can follow. My search for a better GPU based solution failed, and I decided to pursue a CPU based solution instead.

Microsoft used to upkeep a version of R called "Microsoft R Open", yet they discontinued support for it in 2019. Although today's version of R (4.2.2) is significantly better than the most recent Microsoft R Open (4.0.2) version, the Microsoft version has the advantage of using the MKL library to parallelize matrix operation on the CPU, and achieve impressive speedups. The default R program makes the unfortunate mistake of only using one thread from one core at a time, while Microsoft R Open allows you to use multiple (in my case: 8 threads). In addition to this, the MKL package offers optimized versions of the BLAS and LAPACK libraries, which are the libraries that R relies on to perform all of it's linear algebra computation on. Similar to Dr. Nychka's OpenBlas 3.19 replacement for his BLAS, and a default LAPACK, we will instead use these libraries from MKL. It is important to note that this solution will be more effective for those with higher-end CPU's due to them having more cores and threads that can be utilized.

Installation Instructions

A lesson learned from this project is the fact that there are a great many people on the internet that provide computer advice who: enjoy working in the terminal, avoid any form of GUI's, and make tasks out to be much more difficult than they need to be. I am not one of those people. Our installation process is extraordinarily simple, which makes this solution very attractive considering the immense speedup that we will receive. We will download Windows R Open with the MKL option selected, copy the optimized BLAS and LAPACK files from its Program Files, and substitute them for the default files in our most current version of R.

1. Make sure R and Rstudio are closed. Then, go to this website and download the latest version of Microsoft R Open: https://mran.microsoft.com/open
2. Open the downloaded .exe file, and begin the install wizard. When configuring the installation, make sure to select the MKL option as one of the included components! It may already be selected by default, but if this box is not checked, we can not take the necessary optimized libraries.
3. Click through the rest of the install wizard. All other default options should be fine.
4. Now that the program is installed, go to your `(C:)` Drive in your Files. From there, go to `Program Files` -> `Microsoft` -> `R Open` -> `R-4.0.2` -> `bin` -> `x64`.
5. Inside of the `x64` folder, you will see three files: "Rlapack.dll", "Rblas.dll", "libiomp5md.dll". Copy these three files.
6. Now go back to `Program Files`, and then go to your R folder. The path will be `Program Files` -> `R` -> `R-4.2.2` -> `bin` -> `x64`. (Your R version may be different, this is ok.)
7. Delete the default "Rlapack.dll" and "Rblas.dll" files inside of the `x64` folder, and paste in the 3 files that you copied from Microsoft R Open. You will notice that the LAPACK and BLAS files that you pasted in take up significantly more storage space, since they are more complicated than the default ones. In addition, you have also added a third file which was not there previously!

This should be all that you need in order for your R to run significantly faster. For those that are more interested and wish to know how many threads are being used, you must load your R Studio session with R Open, and type in the command `getMKLThreads()`. The number can be modified with `setMKLThreads()` in R Open, but there is no way to check this or change this when you load your R Studio session with your version of R, because we did not transfer all of MKL onto it, so you will have to be content with the numbers you see initially. When you load your RStudio session with your most recent version of R, you will notice that `sessionInfo` does not appear to recognize your optimized libraries, and will say "Matrix products: default". This is ok; you can perform some matrix operations to see that your default R is now significantly faster!

Timing Results

The timing experiments in this report are obtained on my 2021 ROG Zephyrus M16 Laptop, with an Intel i9-11900H processor. As I said previously, these tests are slightly less rigorous than those of the `fieldsMAGMA` report due to me doing homework and having many programs open. Yet again, the raw difference in times for each function is not nearly as important as the speedup factor.

Similar to the tech report, we also perform timing experiments on the `CO2` dataset and a well behaved covariance matrix, and we use the Cholesky decomposition and the `mKrig` functions. Our number of observations varies starting from 200, with the same maximum of 10,000. The sample spatial workflow is replaced by the `spatialProcess` function, which is even more computationally expensive, and in our case it had to perform 20 Cholesky decompositions for maximum likelihood calculations (this can be checked at by looking at the iterations of `$lnLikeEvaluations` in `(name of your spatialProcess)$MLEInfo`). I also included timing experiments for matrix multiplication and finding the inverse of a matrix, because I believe those to be two critical operations in many statistical workflows as well. Graphical results are shown below:

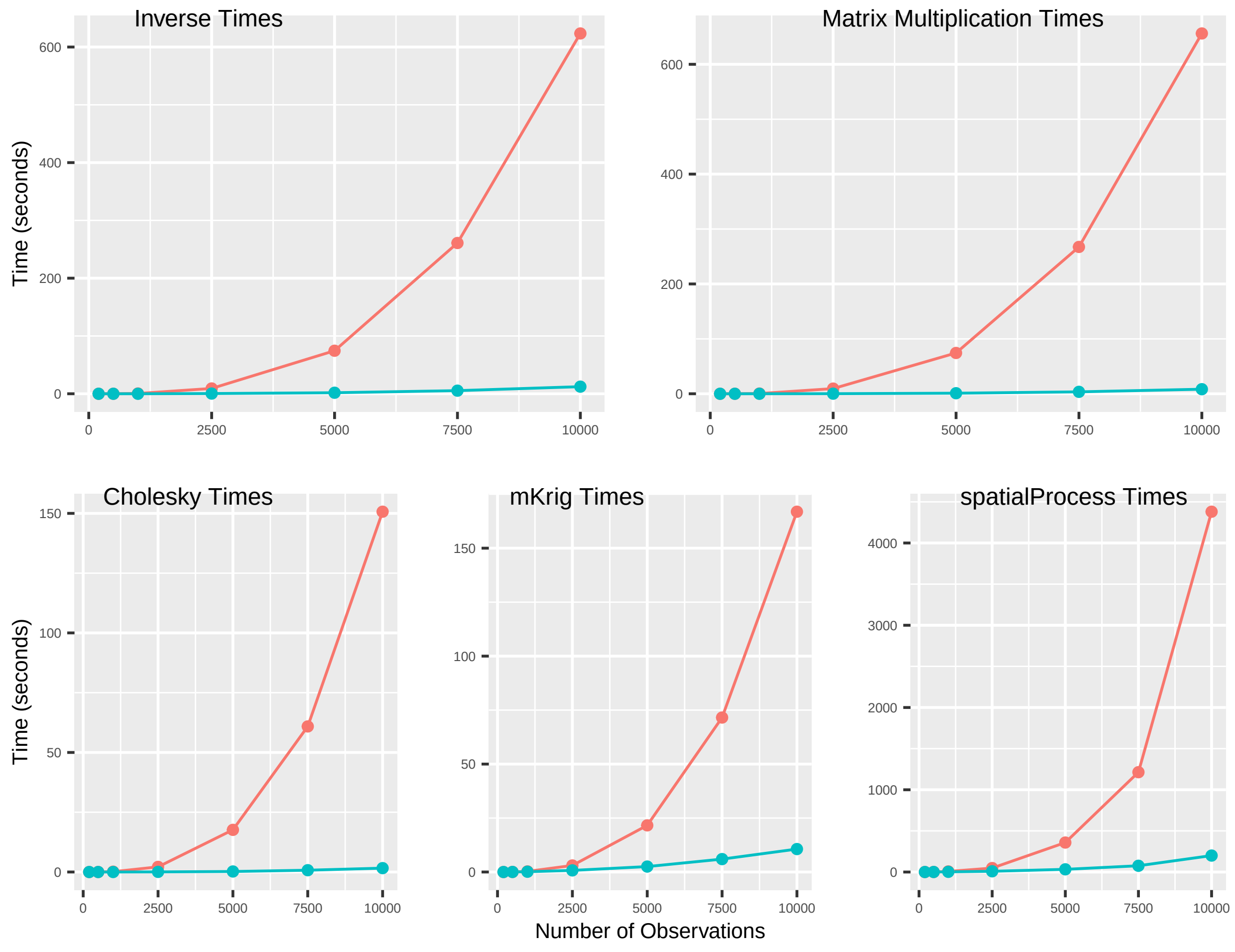

Figure 1: Timing graphs for each function. Red is using default libraries, while Blue is post-speedup.

Yet again, it is important to stress that the timing for the largest number of observations is most important since we aim to approach theoretical computational limits, so I create a table below for the timing values and speedups for 10,000 observations. It is also important to note that while they are rounded in the table, all calculations were done with machine double precision in these tests.

Table 2: Performance Improvements for all Functions Using 10,000 Data Points (MKL CPU Speedup)

Function	Default Time (s)	Speedup Time (s)	Speedup Factor
Inverse	623.5	12.31	50.6
Matrix Multiplication	656.17	8.3	79.1
Cholesky Decomposition (chol)	150.7	1.62	93
Kriging (mKrig)	166.9	10.63	15.7
spatialProcess	4380.60	199.91	21.9

As expected, the default times are slightly slower due to me doing other tasks while performing these tests. On the other hand, the speedup times and factors are incredibly impressive. Again, it is only natural that the more complex functions take slightly longer since they have many other steps within them that do not involve matrix operations. Although the raw time and speedup factor do outperform the `fieldsMAGMA` methods in areas such as Cholesky, and the speedup is greater for the spatial process, it is not entirely safe to compare the results due to too many differing factors.

A final, important thing to check is the accuracy of these new linear algebra libraries. It is only natural for one to be suspicious of such a speedup from simply dragging and dropping files without any cost to the accuracy of the calculations. Although far from a rigorous test, my method was to perform the Cholesky decomposition of the same large 2000 x 2000 matrix before and after the speedup, and find the maximum percent error. At the entry with the maximum difference between the two matrices, the corresponding error percentage of 4.380992e-06%. For all of my purposes, this is an acceptable amount of error. All code for this section, including the timing, tables with all of the entries, plotting, and accuracy check can be found in the Appendix section at the end of this report.

Conclusion and Future Work

This entire project started with my desire to speed up R on my personal device, and in that aspect, I believe I have succeeded. The results obtained from the CPU based solution offer an overall more impressive speedup than those from the `fieldsMAGMA` package, although this may be a result of new technology, and a number of other factors. I suspect that implementing `fieldsMAGMA` on my device today would offer significantly more impressive results than the CPU based solution due to the power of GPUs in modern gaming computers today. Nevertheless, the one advantage that the CPU based solution confidently offers over the GPU solution is it's ease of implementation, especially for Window's users, who seem to have a smaller community when it comes to upgrading the performance of R.

It is quite easy to guess that my future interests are in being able to implement `fieldsMAGMA`, or another GPU based solution onto my device. The `fieldsMAGMA` package is a particularly attractive option, because it effectively provides `fields` users with the same package, except significantly faster right after the download. Two functions are of particular interest: `mKrig` and `spatialProcess`. In the entirety of the `fields` package, these are the only two functions that have to actually use a complete Cholesky decomposition, which is why they are used for timing in both in this and the `fieldsMAGMA` report. Building in faster matrix operations into the `fields` package, or simply finding a way to get them working on Window's device is a related and logical next step for this project. For now, I am content with providing my fellow Window's users with an upgrade that is both significant, and easy to implement.

Appendix (Code)

Timing Script:

This code performs the timing experiments for the inverse of a matrix, matrix multiplication, Cholesky decomposition, `mKrig`, and `spatialProcess` functions. The first three are performed on a synthetic, well behaved random matrix, while the other two functions are performed on shuffled random observations from the 'CO2' dataset that comes with the `fields` package. The shuffling is done on purpose since there are many repeat observations in a row in this dataset that can often cause singularity issues. The third entry from `system.time` is used for the timing because it is the time that is actually experienced by the user staring at the screen, which is what I am most interested in. Length of data varies from 200 to 10,000 observations, resulting in matrices with between 40,000 and 100,000,000 entries in them.

```r
#Libraries (for all code)
suppressMessages(library(fields))
suppressMessages(library(ggplot2))
suppressMessages(library(cowplot))

#Loading in the CO2 data set that comes with fields
data(CO2)
s<- CO2$lon.lat
z<- CO2$y
good<- !is.na( z)
s<- s[good,]
z<- z[good]
N <- length(z)

#setting up our data frame for appending results
#1 run of 1000 to warm up (will be omitted)
datLens <- c(1000, 200, 500, 1000, 2500, 5000, 7500, 10000)
invTimes <- c(NA)
matTimes <- c(NA)
cholTimes <- c(NA)
mKrigTimes <- c(NA)
spatialTimes <- c(NA)

df <- cbind(datLens, invTimes, matTimes, cholTimes, mKrigTimes, spatialTimes)

#setting seed for consistent results
set.seed(777)

for (i in 1:length(datLens)){

  #matrix multiplication timing
  A <- matrix(rnorm((datLens[i])^2), (datLens[i]), (datLens[i]))
  B <- A%*%t(A)
  df[i,2] <- system.time(solve(B))[3]
  df[i,3] <- system.time(B%*%B)[3]

  #cholesky timing
  df[i,4] <- system.time(chol(B))[3]

  #mKrig timing
  IShuffle <- sample( 1:N, datLens[i], replace=FALSE)
  tempS <- s[IShuffle,]
  tempZ <- z[IShuffle]

  #mKrig timing lambda parameter taken from fields documentation
  df[i,5] <- system.time(mKrig(tempS, tempZ, lambda = 0.1))[3]

  #spatialProcess timing
  df[i,6] <- system.time(spatialProcess(tempS, tempZ, lambda = 0.1, smoothness = 0.5))[3]

  #idiot numbers
  cat("Iteration: ", i, "\n")
}

#omitting the first row (warmup run)
df <- df[-1,]
#dfSpeedy <- df

#saving files for different libraries
#save(df, file = "defaultTiming.rda")
#save(dfSpeedy, file = "speedyTiming.rda")
```

Resulting Tables:

The `df` and `dfSpeedy` tables have the timing results for all of the functions. These are not fully displayed in the report, so here are the times for all of the varying numbers of observations:

```
load("defaultTiming.rda")
load("speedyTiming.rda")

print("default times below: ")
```

```
## [1] "default times below: "
```

```
df
```

```
##      datLens invTimes matTimes cholTimes mKrigTimes spatialTimes
## [1,]     200     0.00     0.00      0.00       0.00         0.08
## [2,]     500     0.06     0.05      0.01       0.04         0.59
## [3,]    1000     0.49     0.42      0.14       0.28         7.04
## [4,]    2500     9.20     9.42      2.15       3.03        48.09
## [5,]    5000    74.50    74.33     17.64      21.61       358.25
## [6,]    7500   260.92   267.41     60.89      71.55      1213.55
## [7,]   10000   623.50   656.17    150.70     166.90      4380.60
```

```
print("sped up times below: ")
```

```
## [1] "sped up times below: "
```

```
dfSpeedy
```

```
##      datLens invTimes matTimes cholTimes mKrigTimes spatialTimes
## [1,]     200     0.00     0.00      0.00       0.00         0.07
## [2,]     500     0.01     0.02      0.00       0.03         0.38
## [3,]    1000     0.02     0.02      0.00       0.13         2.22
## [4,]    2500     0.38     0.18      0.05       0.75         8.82
## [5,]    5000     1.81     1.02      0.22       2.52        32.62
## [6,]    7500     5.38     3.51      0.75       5.98        76.05
## [7,]   10000    12.31     8.30      1.62      10.63       199.91
```

Plotting Code:

This code relies heavily on the use of the `cowplot` and `ggplot` libraries to create the image containing all of the graphs in the Timing Results for the CPU based method.

```r
df <- df
dfSpeedy <- dfSpeedy

dfInv <- data.frame(cbind(df[,1], df[,2], dfSpeedy[,2]))
colnames(dfInv) <- c("Run", "OG", "Fast")

dfMat <- data.frame(cbind(df[,1], df[,3], dfSpeedy[,3]))
colnames(dfMat) <- c("Run", "OG", "Fast")

dfChol <- data.frame(cbind(df[,1], df[,4], dfSpeedy[,4]))
colnames(dfChol) <- c("Run", "OG", "Fast")

dfKrig <- data.frame(cbind(df[,1], df[,5], dfSpeedy[,5]))
colnames(dfKrig) <- c("Run", "OG", "Fast")

dfSpat <- data.frame(cbind(df[,1], df[,6], dfSpeedy[,6]))
colnames(dfSpat) <- c("Run", "OG", "Fast")






pInv <- ggplot(dfInv, aes(x = Run)) +
  geom_line(aes(y = OG), color = "#F8766D") +
  geom_line(aes(y = Fast), color = "#00BFC4") +
  geom_point(aes(y = OG), shape = 19, color = "#F8766D") +
  geom_point(aes(y = Fast), shape = 19, color = "#00BFC4") +
  ylab("Time (seconds)") + xlab("") +
  theme(text = element_text(size=rel(2.9)))

pMat <- ggplot(dfMat, aes(x = Run)) +
  geom_line(aes(y = OG), color = "#F8766D") +
  geom_line(aes(y = Fast), color = "#00BFC4") +
  geom_point(aes(y = OG), shape = 19, color = "#F8766D") +
  geom_point(aes(y = Fast), shape = 19, color = "#00BFC4") +
  ylab("") + xlab("") +
  theme(text = element_text(size=rel(2.9)))

topRow <- plot_grid(pInv, pMat,
          labels = c("Inverse Times", "Matrix Multiplication Times"),
          label_size = 9, ncol = 2, label_fontface = "plain",
          label_x = 0.1)

pChol <- ggplot(dfChol, aes(x = Run)) +
  geom_line(aes(y = OG), color = "#F8766D") +
  geom_line(aes(y = Fast), color = "#00BFC4") +
  geom_point(aes(y = OG), shape = 19, color = "#F8766D") +
  geom_point(aes(y = Fast), shape = 19, color = "#00BFC4") +
  xlab("") + ylab("Time (seconds)") +
  theme(text = element_text(size=rel(2.9)))

pKrig <- ggplot(dfKrig, aes(x = Run)) +
  geom_line(aes(y = OG), color = "#F8766D") +
  geom_line(aes(y = Fast), color = "#00BFC4") +
  geom_point(aes(y = OG), shape = 19, color = "#F8766D") +
  geom_point(aes(y = Fast), shape = 19, color = "#00BFC4") +
  xlab("Number of Observations") +
  ylab("") +
  theme(text = element_text(size=rel(2.9)))

pSpat <- ggplot(dfSpat, aes(x = Run)) +
  geom_line(aes(y = OG), color = "#F8766D") +
  geom_line(aes(y = Fast), color = "#00BFC4") +
  geom_point(aes(y = OG), shape = 19, color = "#F8766D") +
  geom_point(aes(y = Fast), shape = 19, color = "#00BFC4") +
  xlab("") + ylab("") +
  theme(text = element_text(size=rel(2.9)))

botRow <- plot_grid(pChol, pKrig, pSpat,
          labels = c("Cholesky Times", " mKrig Times", "spatialProcess Times"),
          label_size = 9, ncol = 3, label_fontface = "plain",
          label_x = 0.05)


timingGraphs <- plot_grid(topRow, botRow, ncol = 1)
timingGraphs
```

Accuracy Checker:

This was the code written to perform the Cholesky decomposition of the same matrix, and compare the difference between the actual values that were obtained before and after the speedup.

```
#Accuracy code
A <- matrix(rnorm(500^2), 500, 500)
B <- A%*%t(A)
choltest <- chol(B)
cholSpeedy <- as.matrix(choltest)


#saving the file to compare to sped up R
#save(choltest, file = "defaultChol.rda")
#save(cholSpeedy, file = "speedyChol.rda")
```

```r
load("defaultChol.rda")
load("speedyChol.rda")

diffMat <- cholSpeedy - choltest
which(diffMat == max(diffMat), arr.ind = TRUE)

#raw maximum error
cat("Maximum Difference between matrices: ", diffMat[500,500], "\n")

#percent error
cat("Maximum Error Percentage: ", diffMat[500,500]/choltest[500,500] * 100, "\n")
#i can live with that
```