Big Data Analytics

Lecture 9: Applied Econometrics with Spark; Machine Learning and GPUs

Prof. Dr. Ulrich Matter
(University of St. Gallen)

30/04/2020

Applied Econometrics with Apache Spark

Spark basics

Building on the MapReduce model discussed in the previous lecture, Apache Spark is a cluster computing platform particularly made for data analytics. From the technical perspective (in very simple terms), Spark improves some shortcomings of the older Hadoop platform, further improving efficiency/speed. More importantly for our purposes, in contrast to Hadoop, Spark is specifically made for analytics tasks and therefore more easily accessible for people with an applied econometrics background but no substantial knowledge in MapReduce cluster computing. In particular, it comes with several high-level operators that make it rather easy to implement analytics tasks. Moreover (in contrast to basic Hadoop), its very easy to use interactively from R, Python, Scala, and SQL. This makes the platform much more accessible and worth the while for empirical economic research, even for relatively simple econometric analyses.

The following figure illustrates the basic components of Spark. The main functionality, including memory management, task scheduling, and the implementation of Spark's capabilities to handle and manipulate data distributed across many nodes in parallel. Several built-in libraries extend the core implementation, covering specific domains of practical data analytics tasks (querying structured data via SQL, processing streams of data, machine learning, and network/graph analysis). The latter two provide various common functions/algorithms frequently used in data analytics/applied econometrics, such as generalized linear regression, summary statistics, and principal component analysis.

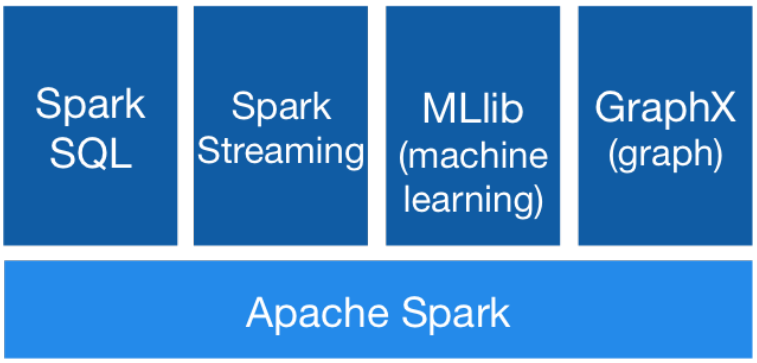


Figure 1: (ref:sparkstack)

(ref:sparkstack) Basic Spark stack (source: https://spark.apache.org/images/spark-stack.png)

At the heart of big data analytics with Spark is the fundamental data structure called 'resilient distributed dataset' (RDD). When loading/importing data into Spark, the data is automatically distributed across the cluster in RDDs (~ as distributed collections of elements) and manipulations are then executed in parallel in

these RDDs. However, the entire Spark framework also works locally on a simple laptop or desktop computer. This is of great advantage when learning Spark and testing/debugging an analytics script on a small sample of the real dataset.

Spark in R

There are two prominent packages to use Spark in connection to R: `SparkR` and RStudio's `sparklyr`, the former is in some ways closer to Spark's Python API, the latter is closer to the `dplyr`-type of data handling (and is 'compatible' with the '`tidyverse`').[1] For the very simple introductory examples below, either package could have been used equally well. For the general introduction we focus on `SparkR` and later have a look at a simple regression example based on `sparklyr`.

To install and use Spark from the R shell, only a few preparatory steps are needed. The following examples are based on installing/running Spark on a Linux machine with the `SparkR` package. `SparkR` depends on Java (version 8). Thus, we first should make sure the right Java version is installed. If several Java versions are installed, we might have to select version 8 manually via the following terminal command (Linux).

With the right version of Java running, we can install `SparkR` as usual with `install.packages("SparkR")`. After installing `SparkR`, the call `SparkR::install.spark()` will download and install Apache Spark to a local directory. No we can start an interactive Spark session from within R.

```
# install.packages("SparkR")

# load packages
library(SparkR)

# start session
sparkR.session()
```

```
## Launching java with spark-submit command /home/umatter/.cache/spark/spark-2.4.5-bin-hadoop2.7/bin/spa
```

```
## Java ref type org.apache.spark.sql.SparkSession id 1
```

By default this starts a local standalone session (no connection to a cluster computer needed). While the examples below are all intended to run on a local machine, it is straightforward to connect to a remote Spark cluster and run the same examples there.[2]

Data import and summary statistics

First, we want to have a brief look at how to perform the first few steps of a typical econometric analysis: import data and compute summary statistics. We analyze the already familiar `flights.csv` dataset. The basic Spark installation provides direct support to import common data formats such as CSV and JSON via the `read.df()` function (for many additional formats, specific Spark libraries are available). To import`flights.csv`, we set the `source`-argument to `"csv"`.

```
# Import data and create a SparkDataFrame (a distributed collection of data, RDD)
flights <- read.df(path="../data/flights.csv", source = "csv", header="true")


# inspect the object
str(flights)
```

```
## 'SparkDataFrame': 19 variables:
##  $ year          : chr "2013" "2013" "2013" "2013" "2013" "2013"
##  $ month         : chr "1" "1" "1" "1" "1" "1"
```

[1] See this blog post for a more detailed comparison and discussion of advantages of either package.

[2] Simply set the `master` argument of `sparkR.session()` to the URL of the Spark master node of the remote cluster. Importantly, the local Spark and Hadoop versions should match the corresponding versions on the remote cluster.

```
##  $ day           : chr "1" "1" "1" "1" "1" "1"
##  $ dep_time      : chr "517" "533" "542" "544" "554" "554"
##  $ sched_dep_time: chr "515" "529" "540" "545" "600" "558"
##  $ dep_delay     : chr "2" "4" "2" "-1" "-6" "-4"
##  $ arr_time      : chr "830" "850" "923" "1004" "812" "740"
##  $ sched_arr_time: chr "819" "830" "850" "1022" "837" "728"
##  $ arr_delay     : chr "11" "20" "33" "-18" "-25" "12"
##  $ carrier       : chr "UA" "UA" "AA" "B6" "DL" "UA"
##  $ flight        : chr "1545" "1714" "1141" "725" "461" "1696"
##  $ tailnum       : chr "N14228" "N24211" "N619AA" "N804JB" "N668DN" "N39463"
##  $ origin        : chr "EWR" "LGA" "JFK" "JFK" "LGA" "EWR"
##  $ dest          : chr "IAH" "IAH" "MIA" "BQN" "ATL" "ORD"
##  $ air_time      : chr "227" "227" "160" "183" "116" "150"
##  $ distance      : chr "1400" "1416" "1089" "1576" "762" "719"
##  $ hour          : chr "5" "5" "5" "5" "6" "5"
##  $ minute        : chr "15" "29" "40" "45" "0" "58"
##  $ time_hour     : chr "2013-01-01T10:00:00Z" "2013-01-01T10:00:00Z" "2013-01-01T10:00:00Z" "2013-01-
```

```
head(flights)
```

```
##   year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay carrier flight
## 1 2013     1   1      517            515         2      830            819        11      UA   1545
## 2 2013     1   1      533            529         4      850            830        20      UA   1714
## 3 2013     1   1      542            540         2      923            850        33      AA   1141
## 4 2013     1   1      544            545        -1     1004           1022       -18      B6    725
## 5 2013     1   1      554            600        -6      812            837       -25      DL    461
## 6 2013     1   1      554            558        -4      740            728        12      UA   1696
##   tailnum origin dest air_time distance hour minute            time_hour
## 1  N14228    EWR  IAH      227     1400    5     15 2013-01-01T10:00:00Z
## 2  N24211    LGA  IAH      227     1416    5     29 2013-01-01T10:00:00Z
## 3  N619AA    JFK  MIA      160     1089    5     40 2013-01-01T10:00:00Z
## 4  N804JB    JFK  BQN      183     1576    5     45 2013-01-01T10:00:00Z
## 5  N668DN    LGA  ATL      116      762    6      0 2013-01-01T11:00:00Z
## 6  N39463    EWR  ORD      150      719    5     58 2013-01-01T10:00:00Z
```

By default, all variables have been imported as type `character`. For several variables this is, of course, not the optimal data type to compute summary statistics. We thus first have to convert some columns to other data types with the `cast` function.

```
flights$dep_delay <- cast(flights$dep_delay, "double")
flights$dep_time <- cast(flights$dep_time, "double")
flights$arr_time <- cast(flights$arr_time, "double")
flights$arr_delay <- cast(flights$arr_delay, "double")
flights$air_time <- cast(flights$air_time, "double")
flights$distance <- cast(flights$distance, "double")
```

Suppose we only want to compute average arrival delays per carrier for flights with a distance over 1000 miles. Variable selection and filtering of observations is implemented in `select()` and `filter()` (as in the `dplyr` package).

```
# filter
long_flights <- select(flights, "carrier", "year", "arr_delay", "distance")
long_flights <- filter(long_flights, long_flights$distance >= 1000)
head(long_flights)
```

```
##   carrier year arr_delay distance
## 1      UA 2013        11     1400
```

```
## 2      UA 2013        20     1416
## 3      AA 2013        33     1089
## 4      B6 2013       -18     1576
## 5      B6 2013        19     1065
## 6      B6 2013        -2     1028
```

Now we summarize the arrival delays for the subset of long flights by carrier. This is the 'split-apply-combine' approach applied in `SparkR`.

```
# aggregation: mean delay per carrier
long_flights_delays<- summarize(groupBy(long_flights, long_flights$carrier),
                      avg_delay = mean(long_flights$arr_delay))
head(long_flights_delays)
```

```
##   carrier  avg_delay
## 1      UA  3.2621897
## 2      AA  0.4957546
## 3      EV 15.6875637
## 4      B6  9.0364413
## 5      DL -0.2393537
## 6      OO -2.0000000
```

Finally, we want to convert the result back into a usual `data.frame` (loaded in our current R session) in order to further process the summary statistics (output to LaTex table, plot, etc.). Note that as in the previous aggregation exercises with the `ff` package, the computed summary statistics (in the form of a table/df) are obviously much smaller than the raw data. However, note that converting a `SparkDataFrame` back into a native R object generally means all the data stored in the RDDs constituting the `SparkDataFrame` object are loaded into local RAM. Hence, when working with actual big data on a Spark cluster, this type of operation can quickly overflow local RAM.

```
# Convert result back into native R object
delays <- collect(long_flights_delays)
class(delays)
```

```
## [1] "data.frame"
```

```
delays
```

```
##    carrier  avg_delay
## 1       UA  3.2621897
## 2       AA  0.4957546
## 3       EV 15.6875637
## 4       B6  9.0364413
## 5       DL -0.2393537
## 6       OO -2.0000000
## 7       F9 21.9207048
## 8       US  0.5566964
## 9       MQ  8.2330896
## 10      HA -6.9152047
## 11      AS -9.9308886
## 12      VX  1.7644644
## 13      WN  9.0841965
## 14      9E  6.6730469
```

Regression analysis

Suppose we want to conduct a correlation study of what factors are associated with more or less arrival delay. Spark provides via its built-in 'MLib' library several high-level functions to conduct regression analyses.

When calling these functions via `SparkR` their usage is actually very similar to the usual R packages/functions commonly used to run regressions in R.

As a simple point of reference, we first estimate a linear model with the usual R approach (all computed in the R environment). First, we load the data as a common `data.table`. We could also convert a copy of the entire `SparkDataFrame` object to a `data.frame` or `data.table` and get essentially the same outcome. However, collecting the data from the RDD structure would take much longer than parsing the csv with `fread`. In addition, we only import the first 300 rows. Running regression analysis with relatively large datasets in Spark on a small local machine might fail or be rather slow.[3]

```
# flights_r <- collect(flights) # very slow!
flights_r <- data.table::fread("../data/flights.csv", nrows = 300)
```

Now we run a simple linear regression (OLS) and show the summary output.

```
# specify the linear model
model1 <- arr_delay ~ dep_delay + distance
# fit the model with ols
fit1 <- lm(model1, flights_r)
# compute t-tests etc.
summary(fit1)
```

```
##
## Call:
## lm(formula = model1, data = flights_r)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -42.386  -9.965  -1.911   9.866  48.024
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.182662   1.676560  -0.109    0.913
## dep_delay    0.989553   0.017282  57.261   <2e-16 ***
## distance     0.000114   0.001239   0.092    0.927
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.49 on 297 degrees of freedom
## Multiple R-squared:  0.9172, Adjusted R-squared:  0.9167
## F-statistic:  1646 on 2 and 297 DF,  p-value: < 2.2e-16
```

Now we aim to compute essentially the same model estimate in `SparkR`. Note that most regression models commonly used in traditional applied econometrics are accessible via `SparkR::glm()`.

```
# create SparkDataFrame
flights2 <- createDataFrame(flights_r)
# fit the model
fit1_spark <- spark.glm( formula = model1, data = flights2 , family="gaussian")
# compute t-tests etc.
summary(fit1_spark)
```

```
##
## Deviance Residuals:
```

[3] Again, it is important to keep in mind that running Spark on a small local machine is only optimal for learning and testing code (based on relatively small samples). The whole framework is not optimized to be run on a small machine but for cluster computers.

```
## (Note: These are approximate quantiles with relative error <= 0.01)
##     Min       1Q   Median       3Q      Max
## -42.386  -10.110   -1.932    9.042   48.024
##
## Coefficients:
##                 Estimate  Std. Error    t value  Pr(>|t|)
## (Intercept)  -0.18266227   1.6765597  -0.108951   0.91332
## dep_delay     0.98955290   0.0172816  57.260607   0.00000
## distance      0.00011396   0.0012385   0.092014   0.92675
##
## (Dispersion parameter for gaussian family taken to be 240.0941)
##
##     Null deviance: 861600  on 299  degrees of freedom
## Residual deviance:  71308  on 297  degrees of freedom
## AIC: 2501
##
## Number of Fisher Scoring iterations: 1
```

Alternative with `sparklyr`:

```
library(sparklyr)

# connect with default configuration
sc <- spark_connect(master = "local",
                    version = "2.4.5")

# load data to spark
flights3 <- copy_to(sc, flights_r, "flights3")

# fit the model
fit2_spark <- ml_linear_regression(flights3, formula = model1)
# compute t-tests etc.
summary(fit2_spark)
```

```
## Deviance Residuals:
##     Min      1Q  Median      3Q     Max
## -42.386  -9.965  -1.911   9.866  48.024
##
## Coefficients:
##   (Intercept)     dep_delay      distance
## -0.1826622687  0.9895529018  0.0001139616
##
## R-Squared: 0.9172
## Root Mean Squared Error: 15.42
```

GPUs for Scientific Computing

The success of the computer games industry in the late 1990s/early 2000s led to an interesting positive externality for scientific computing. The ever more demanding graphics of modern computer games and the huge economic success of the computer games industry set incentives for hardware producers to invest in research and development of more powerful 'graphic cards', extending a normal PC/computing environment with additional computing power solely dedicated to graphics. At the heart of these graphic cards are so-called GPUs (Graphic Processing Units), microprocessors specifically optimized for graphics processing. The image below depicts a modern graphics card with NVIDIA GPUs, which is quite common in today's 'gaming' PCs.

Why did the hardware industry not simply invest in the development of more powerful CPUs to deal with the more demanding PC games? The main reason is that the architecture of CPUs is designed not only for efficiency but also flexibility. That is, a CPU needs to perform well in all kind of computations, some parallel, some sequential, etc. Computing graphics is a comparatively narrow domain of computation and desining a processing unit architecture that is custom-made to excell just at this one task is thus much more cost efficient. Interestingly, this graphics-specific architecture (specialized on highly parallel numerical [floating point] workloads) turns out to be also very usefull in some core scientific computing tasks. In particular, matrix multiplications (see Fatahalian, Sugerman, and Hanrahan (2004) for a detailed discussion of why that is the case). A key aspect of GPUs is that they are composed of several multiprocessor units, of which each has in turn several cores. GPUS thus can perform computations with hundreds or even thousands of threads in parallel. The figure below illustrates this point.

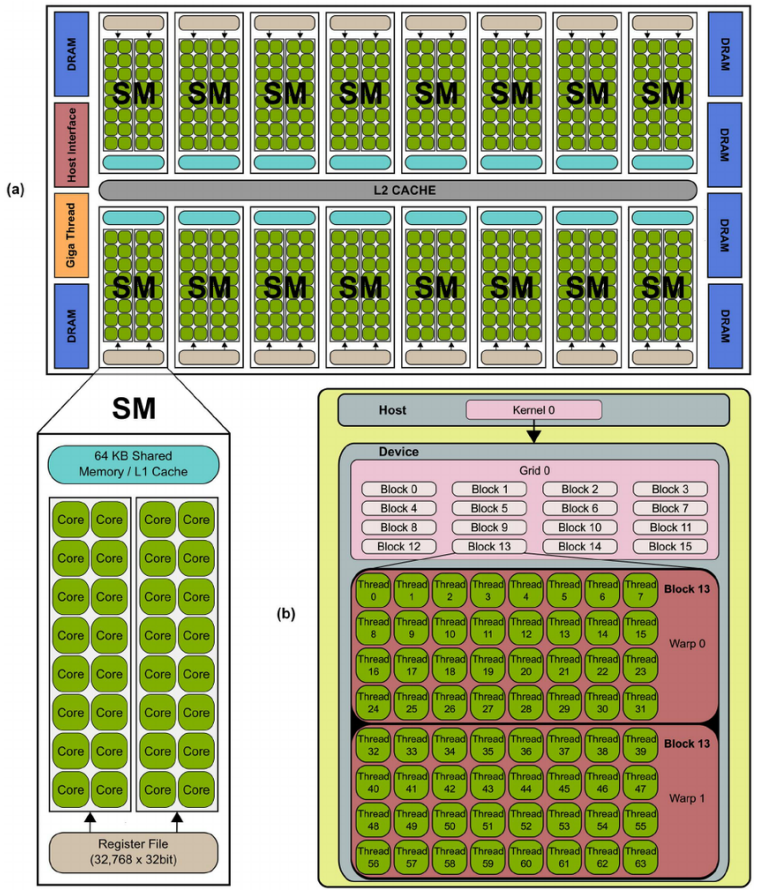


Figure 2: (ref:nvidiaarchitecture)

(ref:nvidiaarchitecture) Typical NVIDIA GPU architecture (illustration and notes by Hernández et al. (2013)): The GPU is comprised of a set of Streaming MultiProcessors (SM). Each SM is comprised of several Stream Processor (SP) cores, as shown for the NVIDIA's Fermi architecture (a). The GPU resources are controlled by the programmer through the CUDA programming model, shown in (b).

While, intially, programming GPUs for scientific computing required a very good understanding of the hardware. Graphics card producers have realized that there is an additional market for their products (in particular with the recent rise of deep learning), and provide several high-level APIs to use GPUs for other tasks than graphics processing. Over the last few years more high-level software has been developed, which makes it much easier to use GPUs in parallel computing tasks. The following subsections shows some examples of such software in the R environment.[4]

GPUs in R

Example I: Matrix multiplication comparison (gpuR)

The `gpuR` package provides basic R functions to compute with GPUs from within the R environmnent. In the following example we compare the performance of the CPU with the GPU based on a matrix multiplication exercise. For a large $N \times P$ matrix X, we want to compute $X^t X$.

In a first step, we load the `gpuR`-package.[5] Note the output to the console. It shows the type of GPU identified by `gpuR`. This is the platform on which `gpuR` will compute the GPU examples. In order to compare the performances, we also load the `bench` package used in previous lectures.

```
# load package
library(bench)
library(gpuR)
```

```
## Number of platforms: 1
## - platform: NVIDIA Corporation: OpenCL 1.2 CUDA 10.2.141
##   - context device index: 0
##     - GeForce GTX 1650
## checked all devices
## completed initialization
```

Next, we initiate a large matrix filled with pseudo random numbers, representing a dataset with N observations and P variables.

```
# initiate dataset with pseudo random numbers
N <- 10000  # number of observations
P <- 100 # number of variables
X <- matrix(rnorm(N * P, 0, 1), nrow = N, ncol =P)
```

For the GPU examples to work, we need one more preparatory step. GPUs have their own memory, which they can access faster than they can access RAM. However, this GPU memory is typically not very large compared to the memory CPUs have access to. Hence, there is a potential trade-off between losing some efficiency but working with more data or vice versa.[6] Here, we show both variants. With `gpuMatrix()` we create an object representing matrix `X` for computation on the GPU. However, this only points the GPU to the matrix and does not actually transfer data to the GPU's memory. The latter is done in the other variant with `vclMatrix()`.

[4]Note that while these examples are easy to implement and run, setting up a GPU for scientific computing still can involve many steps and some knowledge of your computer's system. The examples presuppose that all installation and configuration steps (GPU drivers, CUDA, etc.) have already been completed successfully.

[5]As with the setting up of GPUs on your machine in general, installing all prerequisites to make `gpuR` work on your local machine can be a bit of work and can depend a lot on your system.

[6]If we instruct the GPU to use the own memory, but the data does not fit in it, the program will result in an error.

```
# prepare GPU-specific objects/settings
gpuX <- gpuMatrix(X, type = "float")  # point GPU to matrix (matrix stored in non-GPU memory)
vclX <- vclMatrix(X, type = "float")  # transfer matrix to GPU (matrix stored in GPU memory)
```

Now we run the three examples: first, based on standard R, using the CPU. Then, computing on the GPU but using CPU memory. And finally, computing on the GPU and using GPU memory. In order to make the comparison fair, we force `bench::mark()` to run at least 20 iterations per benchmarked variant.

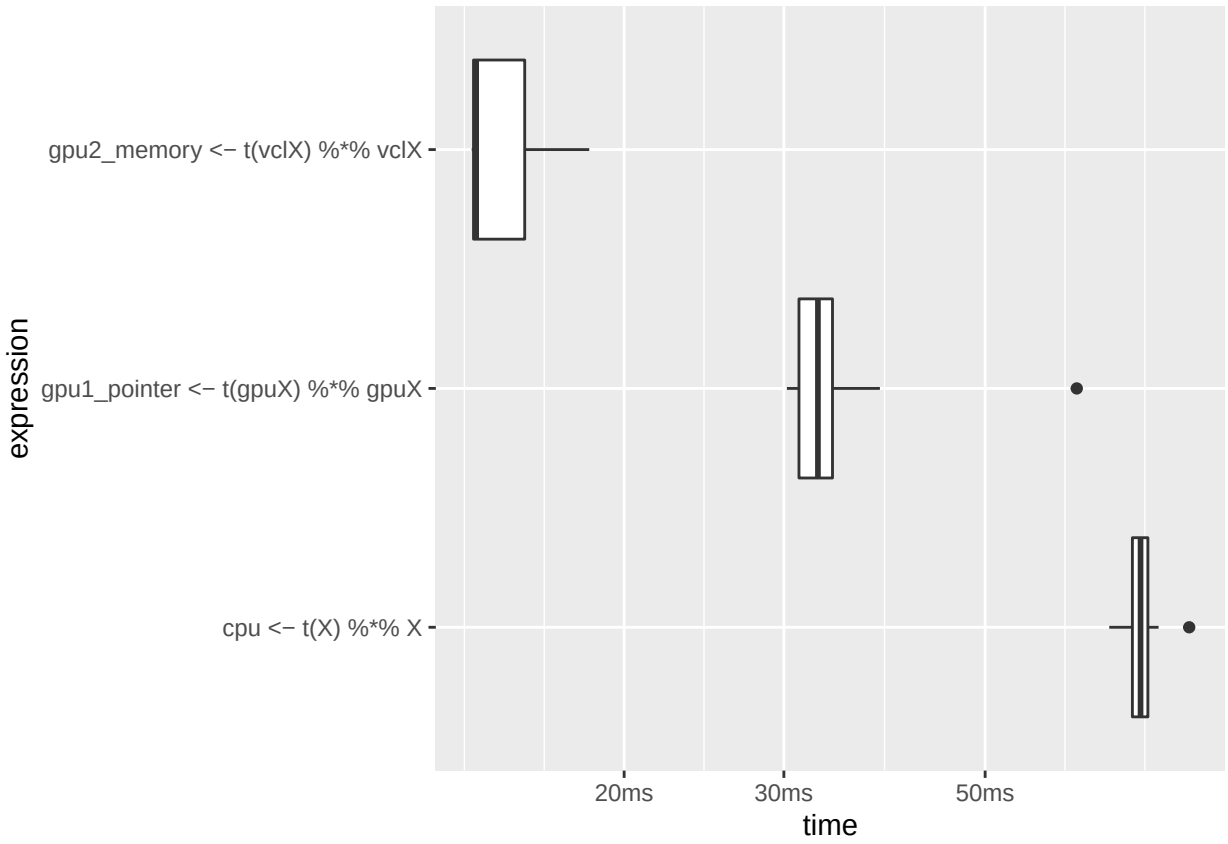
```
# compare three approaches
(gpu_cpu <- bench::mark(

  # compute with CPU
  cpu <- t(X) %*% X,

  # GPU version, GPU pointer to CPU memory (gpuMatrix is simply a pointer)
  gpu1_pointer <- t(gpuX) %*% gpuX,

  # GPU version, in GPU memory (vclMatrix formation is a memory transfer)
  gpu2_memory <- t(vclX) %*% vclX,

check = FALSE, min_iterations = 20))
```

```
## # A tibble: 3 x 6
##   expression                          min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr>                        <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 cpu <- t(X) %*% X                  68.5ms   73.3ms      13.6    7.71MB     4.55
## 2 gpu1_pointer <- t(gpuX) %*% gpuX   30.2ms   32.3ms      29.1  129.62KB     1.53
## 3 gpu2_memory <- t(vclX) %*% vclX    13.6ms   13.7ms      69.3  136.46KB     4.20
```

The performance comparison is visualized with boxplots.

```
plot(gpu_cpu, type = "boxplot")
```

```
## Loading required namespace: tidyr
```



GPUs and Machine Learning

A most common application of GPUs for scientific computing is machine learning, in particular deep learning (machine learning based on artificial neural networks). Training deep learning models can be very computationally intense and to an important part depends on tensor (matrix) multiplications. This is also an area where you might come across highly parallelized computing based on GPUs without even noticing it, as the now commonly used software to build and train deep neural nets (tensorflow, and the high-level Keras API) can easily be run on a CPU or GPU without any further configuration/preparation (apart from the initial installation of these programs). The example below is a simple illustration of how such techniques can be used in an econometrics context.

Tensorflow/Keras example: predict housing prices

In this example we train a simple sequential model with two hidden layers in order to predict the median value of owner-occupied homes (in USD 1,000) in the Boston area (data are from the 1970s). The original data and a detailed description can be found here. The example follows closely this keras tutorial published by RStudio. See RStudio's keras installation guide for how to install keras (and tensorflow) and the corresponding R package `keras`.[7] While the purpose of the example here is to demonstrate a typical (but very simple!) usage case of GPUs in machine learning, the same code should also run on a normal machine (without using GPUs) with a default installation of keras.

Apart from `keras`, we load packages to prepare the data and visualize the output. Via `dataset_boston_housing()`, we load the dataset (shipped with the keras installation) in the format preferred by the `keras` library.

```
# load packages
library(keras)
```

[7] This might involve the installation of additional packages and software outside the R environment.

```
library(tibble)
library(ggplot2)
```

```
##
## Attaching package: 'ggplot2'
```

```
## The following object is masked from 'package:SparkR':
##
##     expr
```

```
# load data
boston_housing <- dataset_boston_housing()
str(boston_housing)
```

```
## List of 2
##  $ train:List of 2
##   ..$ x: num [1:404, 1:13] 1.2325 0.0218 4.8982 0.0396 3.6931 ...
##   ..$ y: num [1:404(1d)] 15.2 42.3 50 21.1 17.7 18.5 11.3 15.6 15.6 14.4 ...
##  $ test :List of 2
##   ..$ x: num [1:102, 1:13] 18.0846 0.1233 0.055 1.2735 0.0715 ...
##   ..$ y: num [1:102(1d)] 7.2 18.8 19 27 22.2 24.5 31.2 22.9 20.5 23.2 ...
```

In a first step, we split the data into a training set and a test set. The latter is used to monitor the out-of-sample performance of the model fit. Testing the validity of an estimated model by looking at how it performs out-of-sample is of particular relevance when working with (deep) neural networks, as they can easily lead to over-fitting. Validity checks based on the test sample are, therefore, often an integral part of modelling with tensorflow/keras.

```
# assign training and test data/labels
c(train_data, train_labels) %<-% boston_housing$train
c(test_data, test_labels) %<-% boston_housing$test
```

In order to better understand and interpret the dataset we add the original variable names, and convert it to a `tibble`.

```
# prepare data
column_names <- c('CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE',
                  'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT')
train_df <- as_tibble(train_data)
colnames(train_df) <- column_names

train_df
```

```
## # A tibble: 404 x 13
##      CRIM    ZN INDUS  CHAS   NOX    RM   AGE   DIS   RAD   TAX PTRATIO     B LSTAT
##     <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>   <dbl> <dbl> <dbl>
##  1 1.23     0    8.14     0 0.538  6.14  91.7  3.98     4   307    21    397. 18.7
##  2 0.0218  82.5  2.03     0 0.415  7.61  15.7  6.27     2   348    14.7  395.  3.11
##  3 4.90     0   18.1      0 0.631  4.97 100    1.33    24   666    20.2  376.  3.26
##  4 0.0396   0    5.19     0 0.515  6.04  34.5  5.99     5   224    20.2  397.  8.01
##  5 3.69     0   18.1      0 0.713  6.38  88.4  2.57    24   666    20.2  391. 14.6
##  6 0.284    0    7.38     0 0.493  5.71  74.3  4.72     5   287    19.6  391. 11.7
##  7 9.19     0   18.1      0 0.7    5.54 100    1.58    24   666    20.2  397. 23.6
##  8 4.10     0   19.6      0 0.871  5.47 100    1.41     5   403    14.7  397. 26.4
##  9 2.16     0   19.6      0 0.871  5.63 100    1.52     5   403    14.7  169. 16.6
## 10 1.63     0   21.9      0 0.624  5.02 100    1.44     4   437    21.2  397. 34.4
## # ... with 394 more rows
```

Next, we have a close look at the data. Note the usage of the term 'label' for what is usually called the 'dependent variable' in econometrics.[8] As the aim of the exercise is to predict median prices of homes, the output of the model will be a continuous value ('labels').

```
# check example data dimensions and content
paste0("Training entries: ", length(train_data), ", labels: ", length(train_labels))
```

```
## [1] "Training entries: 5252, labels: 404"
```

```
summary(train_data)
```

```
##        V1                  V2                V3              V4                V5
##  Min.   : 0.00632   Min.   :  0.00   Min.   : 0.46   Min.   :0.00000   Min.   :0.3850
##  1st Qu.: 0.08144   1st Qu.:  0.00   1st Qu.: 5.13   1st Qu.:0.00000   1st Qu.:0.4530
##  Median : 0.26888   Median :  0.00   Median : 9.69   Median :0.00000   Median :0.5380
##  Mean   : 3.74511   Mean   : 11.48   Mean   :11.10   Mean   :0.06188   Mean   :0.5574
##  3rd Qu.: 3.67481   3rd Qu.: 12.50   3rd Qu.:18.10   3rd Qu.:0.00000   3rd Qu.:0.6310
##  Max.   :88.97620   Max.   :100.00   Max.   :27.74   Max.   :1.00000   Max.   :0.8710
##        V6              V7               V8              V9              V10             V11
##  Min.   :3.561   Min.   :  2.90   Min.   : 1.130   Min.   : 1.000   Min.   :188.0   Min.   :12.60
##  1st Qu.:5.875   1st Qu.: 45.48   1st Qu.: 2.077   1st Qu.: 4.000   1st Qu.:279.0   1st Qu.:17.23
##  Median :6.199   Median : 78.50   Median : 3.142   Median : 5.000   Median :330.0   Median :19.10
##  Mean   :6.267   Mean   : 69.01   Mean   : 3.740   Mean   : 9.441   Mean   :405.9   Mean   :18.48
##  3rd Qu.:6.609   3rd Qu.: 94.10   3rd Qu.: 5.118   3rd Qu.:24.000   3rd Qu.:666.0   3rd Qu.:20.20
##  Max.   :8.725   Max.   :100.00   Max.   :10.710   Max.   :24.000   Max.   :711.0   Max.   :22.00
##       V12             V13
##  Min.   :  0.32   Min.   : 1.73
##  1st Qu.:374.67   1st Qu.: 6.89
##  Median :391.25   Median :11.39
##  Mean   :354.78   Mean   :12.74
##  3rd Qu.:396.16   3rd Qu.:17.09
##  Max.   :396.90   Max.   :37.97
```

```
summary(train_labels) # Display first 10 entries
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    5.00   16.68   20.75   22.40   24.80   50.00
```

As the dataset contains variables ranging from per capita crime rate to indicators for highway access, the variables are obviously measured in different units and hence displayed on different scales. This is not per se a problem for the fitting procedure. However, fitting is more efficient when all features (variables) are normalized.

```
# Normalize training data
train_data <- scale(train_data)

# Use means and standard deviations from training set to normalize test set
col_means_train <- attr(train_data, "scaled:center")
col_stddevs_train <- attr(train_data, "scaled:scale")
test_data <- scale(test_data, center = col_means_train, scale = col_stddevs_train)

train_data[1, ] # First training sample, normalized
```

```
##  [1] -0.2719092 -0.4830166 -0.4352220 -0.2565147 -0.1650220 -0.1762241  0.8120550  0.1165538
##  [9] -0.6254735 -0.5944330  1.1470781  0.4475222  0.8241983
```

[8] Typical textbook examples in machine learning deal with classification (e.g. a logit model), while in microeconometrics the typical example is usually a linear model (continuous dependent variable).

We specify the model as a linear stack of layers: The input (all 13 explanatory variables), two densely connected hidden layers (each with a 64-dimensional output space), and finally the one-dimensional output layer (the 'dependent variable').

```
# Create the model
# model specification
model <- keras_model_sequential() %>%
  layer_dense(units = 64, activation = "relu",
              input_shape = dim(train_data)[2]) %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 1)
```

In order to fit the model, we first have to compile it (configure it for training). At this step we set the configuration parameters that will guide the training/optimization procedure. We use the mean squared errors loss function (`mse`) typically used for regressions. We chose the RMSProp optimizer to find the minimum loss.

```
# compile the model
  model %>% compile(
    loss = "mse",
    optimizer = optimizer_rmsprop(),
    metrics = list("mean_absolute_error"))
```

Now we can get a summary of the model we are about to fit to the data.

```
# get a summary of the model
model
```

```
## Model
## Model: "sequential"
## _________________________________________________________________________________________________
## Layer (type)                                Output Shape                              Param #
## =================================================================================================
## dense (Dense)                               (None, 64)                                896
## _________________________________________________________________________________________________
## dense_1 (Dense)                             (None, 64)                                4160
## _________________________________________________________________________________________________
## dense_2 (Dense)                             (None, 1)                                 65
## =================================================================================================
## Total params: 5,121
## Trainable params: 5,121
## Non-trainable params: 0
## _________________________________________________________________________________________________
```

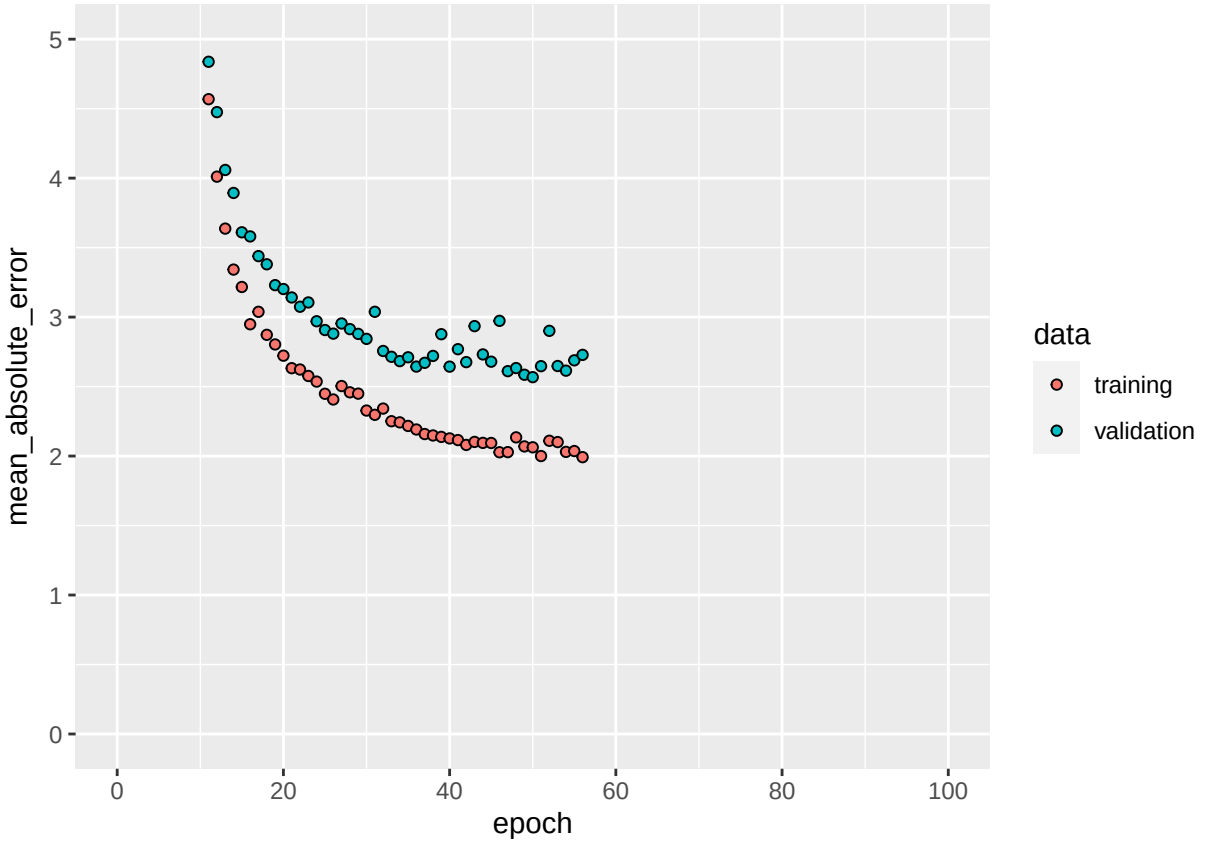
Given the relatively simple model and small dataset, we set the maximum number of epochs to 500 and allow for early stopping in case the validation loss (based on test data) is not improving for a while.

```
# Set max. number of epochs
epochs <- 500

# The patience parameter is the amount of epochs to check for improvement.
early_stop <- callback_early_stopping(monitor = "val_loss", patience = 20)
```

Finally, we fit the model while preserving the training history, and visualize the training progress.

```
# Fit the model and store training stats
history <- model %>% fit(
  train_data,
  train_labels,
```

```
  epochs = epochs,
  validation_split = 0.2,
  verbose = 0,
  callbacks = list(early_stop)
)

plot(history, metrics = "mean_absolute_error", smooth = FALSE) +
  coord_cartesian(xlim = c(0, 100), ylim = c(0, 5))
```



A word of caution

From just comparing the number of threads of a modern CPU with the number of threads of a modern GPU, one might get the impression that parallel tasks should always be implemented for GPU computing. However, whether one approach or the other is faster can depend a lot on the overall task and the data at hand. Moreover, the parallel implementation of tasks can be done more or less well on either system. Really efficient parallel implementation of tasks can take a lot of coding time (particularly when done for GPUs).[9].

References

Fatahalian, K., J. Sugerman, and P. Hanrahan. 2004. "Understanding the Efficiency of Gpu Algorithms for Matrix-Matrix Multiplication." In *Proceedings of the Acm Siggraph/Eurographics Conference on Graphics Hardware*, 133–37. HWWS '04. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1058129.1058148.

[9]For a more detailed discussion of the relevant factors for well done parallelization (either on CPUs or GPUs), see Matloff (2015)

Hernández, Moisés, Ginés D. Guerrero, José M. Cecilia, José M. García, Alberto Inuggi, Saad Jbabdi, Timothy E. J. Behrens, and Stamatios N. Sotiropoulos. 2013. "Accelerating Fibre Orientation Estimation from Diffusion Weighted Magnetic Resonance Imaging Using Gpus." *PLOS ONE* 8 (4): 1–13. https://doi.org/10.1371/journal.pone.0061892.

Matloff, Norman. 2015. *Parallel Computing for Data Science.* Boca Raton, FL: CRC Press.