Terapixel Project

Michael Luke Battle

30/12/2020

Introduction

Business Understanding

Due to the ever-expanding amount of data produced by cities, it is becoming increasingly difficult to effectively convey this information to stakeholders. A possible solution to this problem, devised by Newcastle University, is to use terapixel images. These images consist of over one trillion pixels, allowing the user to interactively browse big data. To highlight the potential of terapixel images, Newcastle University have created a terapixel image of the city of Newcastle-upon-Tyne. Not only does this image provide an interactive visualisation of the city, but it also includes data taken from sensors in various locations describing the temperature and humidity distribution. The creation of such an image illustrates the feasibility of terapixel images from a technical standpoint, whilst simultaneously providing new and exciting routes of exploration in the field of smart cities.

However, rendering over one trillion pixels requires immense computing power, and so the University has made use of public cloud service *Microsoft Azure*, to perform the rendering process. The use of this IaaS enables two key objectives of the project to be fulfilled. Firstly, it facilitates scaling of the visualisation by simply adding more nodes to the cloud supercomputer. Within the context of the project, this is important should more data from sensors be added, or the Newcastle area depicted in the image require expansion. Secondly, this allows for a much quicker rendering process, which ensures that the terapixel image can support daily updates. This is an vital feature to ensure that sensor data remains up-to-date.

Whilst Newcastle University has created this terapixel visualisation, it is important that the rendering process with cloud supercomputing is rigorously evaluated in order to identify any inefficiencies. The aims of this analysis will be to thoroughly evaluate individual events performed as part of the total rendering process, explore the relationship between different performance metrics, identify GPU cards and tiles with unusual behavior and assess the efficiency of the task scheduling process.

Data Understanding

Describe Data

Newcastle University has provided three "csv" files detailing various aspects of a single, entire run of the rendering process. The file "application.checkpoints.csv" has values all of class character, with dimensions and variable names:

```
dim(application.checkpoints)
```

```
## [1] 660400      6
```

```
colnames(application.checkpoints)
```

```
## [1] "timestamp" "hostname"  "eventName" "eventType" "jobId"     "taskId"
```

```
str(application.checkpoints)
```

```
## tibble [660,400 x 6] (S3: tbl_df/tbl/data.frame)
##  $ timestamp: chr [1:660400] "2018-11-08T07:41:55.921Z" "2018-11-08T07:42:29.842Z" "2018-11-08T07:42
##  $ hostname : chr [1:660400] "0d56a730076643d585f77e00d2d8521a00000N" "0d56a730076643d585f77e00d2d851
##  $ eventName: chr [1:660400] "Tiling" "Saving Config" "Saving Config" "Render" ...
##  $ eventType: chr [1:660400] "STOP" "START" "STOP" "START" ...
##  $ jobId    : chr [1:660400] "1024-lvl12-7e026be3-5fd0-48ee-b7d1-abd61f747705" "1024-lvl12-7e026be3-5
##  $ taskId   : chr [1:660400] "b47f0263-ba1c-48a7-8d29-4bf021b72043" "20fb9fcf-a927-4a4b-a64c-70258b60
```

Where "eventType" can be either "START" or "STOP" to indicate the end of beginning of an event. The "eventName" variable details the events that make up the entire rendering task per tile. This variable has the following values;

```
unique(application.checkpoints$eventName)
```

```
## [1] "Tiling"        "Saving Config" "Render"        "TotalRender"
## [5] "Uploading"
```

In addition to "eventType" and "eventName", "timestamp" gives the time associated with this event starting or stopping, "hostname" is the unique hostname of the virtual machine auto-assigned by the Azure batch system. "jobId" and "taskId" are the unique IDs of the Azure batch job and task, respectively.

The file "gpu" has the following dimensions and variable names:

```
dim(gpu)
```

```
## [1] 1543681       8
```

```
colnames(gpu)
```

```
## [1] "timestamp"      "hostname"       "gpuSerial"      "gpuUUID"
## [5] "powerDrawWatt"  "gpuTempC"       "gpuUtilPerc"    "gpuMemUtilPerc"
```

The variable names "timestamp" and "hostname" have the same meaning as in "application.checkpoints". "gpuSerial" and "gpuUUID" are the serial number and unique ID of the physical GPU card, representing a one-to-one match the "hostname" variable. As there are 1024 gpu cores in the cloud supercomputer, there are 1024 different values for "hostname", "gpuSerial" and "gpuUUID".

```
length(unique(gpu$hostname))
```

```
## [1] 1024
```

```
length(unique(gpu$gpuSerial))
```

```
## [1] 1024
```

```
length(unique(gpu$gpuUUID))
```

```
## [1] 1024
```

```
dim(unique(gpu[c("hostname","gpuSerial","gpuUUID")]))
```

```
## [1] 1024    3
```

Other variables in this data set are numeric variables detailing various performance metrics of the GPU cores. These include "powerDrawWatt" (the power draw of the GPU in watts), "gpuTempC" (the temperature of the GPU in celsius), "gpuUtilPerc" (the percent utilisation of the GPU core) and "gpuMemUtilPerc" (the percent utilisation of the GPU memory).

We can also observe that variables "timestamp", "hostname" and "gpuUUID" are of class character, whilst variables "gpuSerial" and "powerDrawWatt" are of class numeric. Lastly, variables "gpuTempC", "gpuUtilPerc" and "gpuMemUtilPerc" are integer values.

```
str(gpu)
```

```
## tibble [1,543,681 x 8] (S3: tbl_df/tbl/data.frame)
##  $ timestamp     : chr [1:1543681] "2018-11-08T08:27:10.314Z" "2018-11-08T08:27:10.192Z" "2018-11-08
##  $ hostname      : chr [1:1543681] "8b6a0eebc87b4cb2b0539e81075191b900001C" "d8241877cd994572b46c861
##  $ gpuSerial     : num [1:1543681] 3.23e+11 3.24e+11 3.23e+11 3.25e+11 3.23e+11 ...
##  $ gpuUUID       : chr [1:1543681] "GPU-1d1602dc-f615-a7c7-ab53-fb4a7a479534" "GPU-04a2dea7-f4f1-12d
##  $ powerDrawWatt : num [1:1543681] 131.6 117 121.6 50.2 141.8 ...
##  $ gpuTempC      : int [1:1543681] 48 40 45 38 41 43 41 35 43 36 ...
##  $ gpuUtilPerc   : int [1:1543681] 92 92 91 90 90 88 91 0 93 90 ...
##  $ gpuMemUtilPerc: int [1:1543681] 53 48 44 43 47 40 47 0 56 40 ...
```

The final file that we will use is "task.x.y". This file has the dimensions and variables:

```
dim(task.x.y)
```

```
## [1] 65793     5
```

```
colnames(task.x.y)
```

```
## [1] "taskId" "jobId"  "x"      "y"      "level"
```

Variables "jobId" and "taskId" are the same as in "application.checkpoints", whilst "x" and "y" represent the location of each tile being rendered. The "level" variable allows users to zoom into the visualisation. In total there are 12 levels, however only levels 12, 8 and 4 are rendered whilst the other levels are derived in the "tiling" process. Level 12 is when the the image is at maximum zoom, and level 1 is when the image is fully zoomed out. The "jobId" relates to the "level" variable, so in total there are three unique "jobId" values, corresponding to the three levels that are rendered, as shown below:

```
unique(task.x.y[c("jobId","level")])
```

```
## # A tibble: 3 x 2
##   jobId                                        level
##   <chr>                                        <int>
## 1 1024-lvl12-7e026be3-5fd0-48ee-b7d1-abd61f747705    12
## 2 1024-lvl4-90b0c947-dcfc-4eea-a1ee-efe843b698df      4
## 3 1024-lvl8-5ad819e1-fbf2-42e0-8f16-a3baca825a63      8
```

We can also observe that the values in "taskId" and "jobId" are of class character, whilst the other variables are of class integer.

```
str(task.x.y)
```

```
## tibble [65,793 x 5] (S3: tbl_df/tbl/data.frame)
##  $ taskId: chr [1:65793] "00004e77-304c-4fbd-88a1-1346ef947567" "0002afb5-d05e-4da9-bd53-7b6dc19ea6d4
##  $ jobId : chr [1:65793] "1024-lvl12-7e026be3-5fd0-48ee-b7d1-abd61f747705" "1024-lvl12-7e026be3-5fd0-
##  $ x     : int [1:65793] 116 142 142 235 171 179 255 218 241 22 ...
##  $ y     : int [1:65793] 178 190 86 11 53 226 61 250 166 220 ...
##  $ level : int [1:65793] 12 12 12 12 12 12 12 12 12 12 ...
```

Data Quality

With the data described at a high level, we can check for any missing values using the below code.

```
sum(is.na(application.checkpoints))
```

```
## [1] 0
```

```
sum(is.na(gpu))
```

```
## [1] 0
```

```
sum(is.na(task.x.y))
```

```
## [1] 0
```

Therefore, there is no missing data. We can now check the data for duplicates.

```
dim(application.checkpoints)[1] - dim(unique(application.checkpoints))[1]
```

```
## [1] 2470
```

```
dim(gpu)[1] - dim(unique(gpu))[1]
```

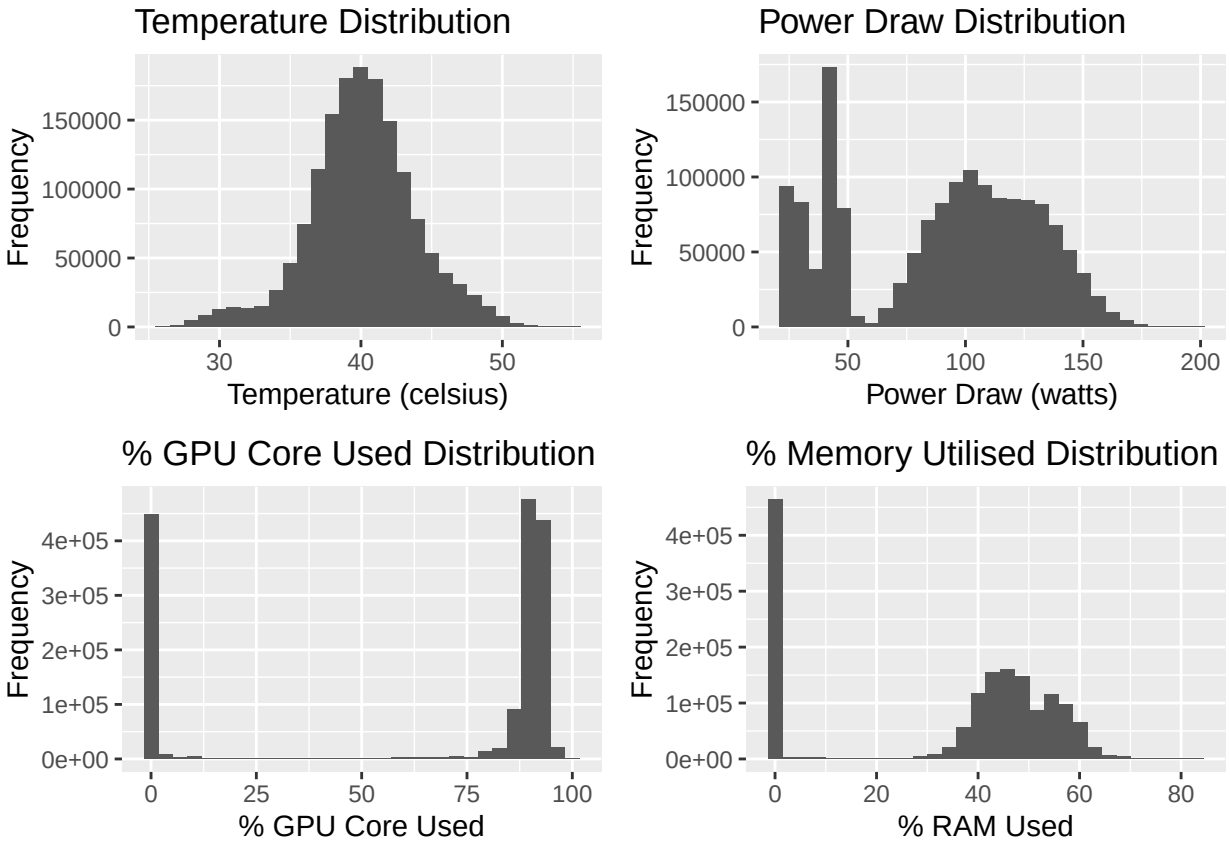
```
## [1] 9
```

```
dim(task.x.y)[1] - dim(unique(task.x.y))[1]
```

```
## [1] 0
```

We can see that there are 2,470 duplicates in "application.checkpoints", 9 in "gpu" and none in "task.x.y.csv". With no missing data and only a small percentage of duplicate data, the data is of good quality.

We can gain some high-lvel insight into the numeric variables in "gpu" by plotting histograms for each of these variables.



From the above histograms we can can observe significant noise in variables gpuUtilPerc, gpuMemUtilPerc and powerDrawWatt. This noise is localised around 0 in each case. In the subsequent section these outliers will be investigated and the data transformed into the a more suitable format for analysis.

Data Preparation

To ensure that the evaluation of the supercomputer rendering process is rigorous, it is important to examine all aspects of it. We will therefore use the data in all three files for our analysis. First of all, we will prepare the data in "application.checkpoints".

First, to make the "timestamp" variable useful in calculations, we can convert it from a character format to datetime, or POSIXct. As each value in "timestamp" is of the form "2018-11-08T07:41:55.921Z", we can use the custom function **clean_date_time()** to remove the letters "T" and "Z". All custom functions detailed in this report are located in "helpers.R", within the "lib" project sub-directory.

The resulting "timestamp" values are parsed into into POSIXct form, before the 2,470 duplicates found in the *Data Understanding* section are removed using the **unique()** function.

With the "timestamp" variable in a more computation-friendly format, we can now derive the total runtime for each event. First, the data must be converted from long format to wide format with respect to the "eventType" variable. Note that the data is also arranged by hostname and START time, a step that will prove useful in our future data preparation. Then, the start time of the event can be subtracted from the stop time to calculate the runtime in seconds. This is performed with the below code.

```
#convert to wide format so that START and STOP are separate columns
app_wide = app_data %>%
  pivot_wider(names_from = eventType,
              values_from = timestamp) %>%
  arrange(hostname,START)

#create runtime variable by calculating time difference from start to stop timestamp
app_wide$runtime = as.numeric(app_wide$STOP - app_wide$START)
```

Using "runtime", we can check if the "TotalRender" runtime matches the summation of the runtimes for the other events by further widening the data to create a variable for each eventName, whilst also excluding variables "START" and "STOP" such that each observation in the data corresponds to an individual task. We will do this in a different data set called "event_data" to preserve the data pipeline which "app_wide" is integral to. A new variable, "TotalProcess", is then created which is the summation of the runtimes for events "Saving Config", "Render", "Tiling" and "Uploading". Creating a new variable, "diff", to calculate the difference between the original "TotalRender" variable and"TotalProcess", reveals that the runtime for "Tiling" is not captured in the "TotalRender" variable in the majority of cases.

```
event_data = app_wide[c("eventName","runtime","taskId","hostname")] %>%
  arrange(taskId) %>%
  pivot_wider(names_from = eventName,
              values_from = runtime)

event_data$totalprocess = event_data$Render + event_data$Uploading +
  event_data$`Saving Config` + event_data$Tiling

event_data$diff = event_data$totalprocess - event_data$TotalRender
```

We can then confirm that the difference between the "TotalRender" and "TotalProcess" variables is almost identical to the runtime of "Tiling" with the below calculation.

```
sum(as.numeric(event_data$diff))/sum(as.numeric(event_data$Tiling))
```

```
## [1] 0.9995459
```

However, a look at the timeline of event executions for a single task reveals that the "Uploading" and "Tiling" events are initialised concurrently.

```
app_wide[app_wide$taskId == unique(app_wide$taskId)[1],
         c("eventName","START","STOP","runtime")]
```

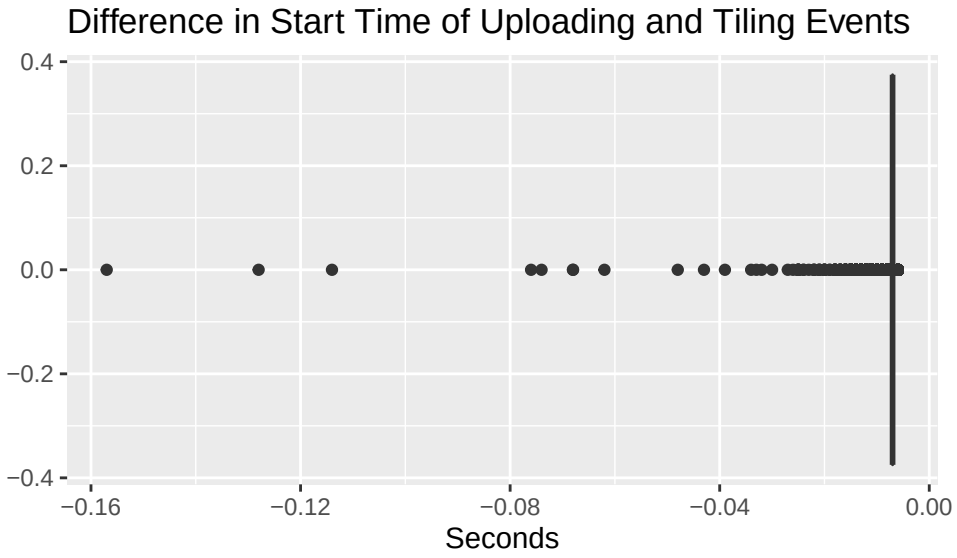
```
## # A tibble: 5 x 4
##   eventName     START               STOP                runtime
##   <chr>         <dttm>              <dttm>                <dbl>
## 1 TotalRender   2018-11-08 07:41:45 2018-11-08 07:42:11 25.8
## 2 Saving Config 2018-11-08 07:41:45 2018-11-08 07:41:45  0.00200
```

```
## 3 Render         2018-11-08 07:41:45 2018-11-08 07:42:09 23.9
## 4 Uploading      2018-11-08 07:42:09 2018-11-08 07:42:11  1.92
## 5 Tiling         2018-11-08 07:42:09 2018-11-08 07:42:10  0.723
```

Due to this, the runtime of "TotalRender" will not be equal to the summation of the runtimes for the other events. To verify that "Uploading" and "Tiling" are initialised at the same time for all rendering tasks in the file, we can widen the data to view only these events and their respective start time for each task. By then adding a new column that calculates the difference between each start time, we compute a vector of all differences, "event_start_diff".

```
event_diff = filter(app_wide,eventName %in% c("Uploading","Tiling")) %>%
  select(eventName,taskId,START) %>%
  pivot_wider(names_from = eventName,
              values_from = START) %>%
  mutate(start_diff = Uploading - Tiling)

event_start_diff = as.numeric(event_diff$start_diff)
```

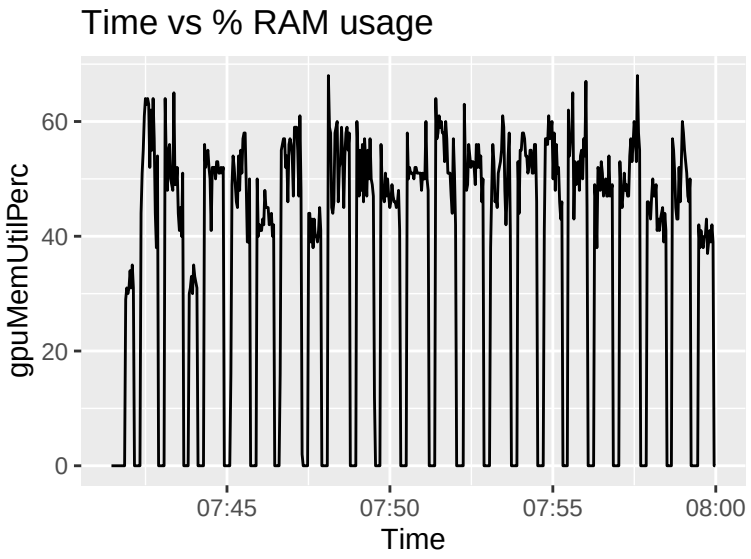
In viewing the distribution of this difference vector in the below boxplot, it is clear that the differences between the start times for "Uploading" and "Tiling" are negligible. The largest difference between the two start times over the whole data is around 0.16 seconds, while the majority of differences are distributed heavily around 0.01 seconds. Therefore, it is fair to assume that they are initialised at the same time for every task.



In order to view the interaction of the "runtime" variable with variables from other data sets, it would be useful to create a new data set that exclusively contains eventName "TotalRender", thus showing only the total runtime for each rendering task per virtual machine. To further enhance this data set and make it conducive to joining with the "gpu" data later in the analysis, a "task_no" variable will also be added. This reflects the chronological order that each rendering task is performed per virtual machine. Creating this new variable uses custom function **assign_tr_task()**, which is applied to a list of the unique hostnames in the data. This results in a list of lists, which is unpacked and appended to the pre-ordered data. These steps are performed using the following code:

```
totalrender_data = filter(app_wide,eventName == "TotalRender")

unique_app_hostname = unique(totalrender_data$hostname)

task_no_per_host = lapply(unique_app_hostname,assign_tr_task)

task_no_per_host = unlist(task_no_per_host)

totalrender_data$task_no = task_no_per_host
```

There are several issues with the data in "gpu" that we must address. Firstly, the values in the "timestamp" variable are in the same class and format as in the raw "application.checkpoints" file. As previously with "application.checkpoints", we can transform this into a more meaningful and computationally compatible variable. We should also remove the aforementioned duplicates from the data. With the "timestamp" values now of class POSIXct and duplicates removed, we can turn our attention to the outliers in the data. Whether these outliers should be retained or not is dependent upon how meaningful they are. We can now arrange "gpu" by "hostname" and the newly transformed variable "timestamp". The resulting data provides several new insights into the "gpu" data. Firstly, it shows that the "gpu" data is a series of a snapshot measurements for each virtual machine taken every 2 seconds.Secondly, it indicates that the observations where gpuMemUtilPerc have a value of 0 are related to the periods on the virtual machine in between tasks. This can be best depicted using a plot of gpuMemUtilPerc vs Time on a single virtual machine.



In the above plot it is evident that the peaks correspond to when a task is performed on the virtual machine and computer memory is being used. For analysis on the performance metrics whilst a task is being performed, any observations with a value of 0 for the "gpuMemUtilPerc" and "gpuUtilPerc" variables should be removed. However, for an investigation into the efficiency of the task scheduling process we can use these observations to calculate the amount of time the virtual machine spends in between tasks.

To facilitate combining all three data sets, we will create the variable "task_no" in "gpu_data" that will assign a task number to each observation within a single peak in the above plot, such that the number of unique task numbers corresponds to the number of peaks for each virtual machine. This is performed for each unique hostname using the custom function **assign_task_no()**. This function identifies a new task, or peak, and assigns the corresponding task number when several criteria have been satisfied. These are if the previous two values in "gpuMemUtilPerc" are 0, and both the "gpuMemUtilPerc" and "gpuUtilPerc"

values in the current row are non-zero. Once the "task_no" variable has been added to the data, we create a new data set called "gpu_summary". This groups the data by the hostname and task_no, then summarises the performance metrics for each task with the arithmetic mean. The code for the steps described above is shown below.

```
unique_hostnames = unique(gpu_data$hostname)

task_no_gpu = lapply(unique_hostnames,assign_task_no)

task_no_gpu = unlist(task_no_gpu)

gpu_data$task_no = task_no_gpu

gpu_summary = gpu_data[gpu_data$gpuMemUtilPerc!=0,] %>%
  group_by(hostname,task_no,gpuSerial) %>%
  summarise(powerDraw = mean(powerDrawWatt),
            tempC = mean(gpuTempC),
            MemUtilPerc = mean(gpuMemUtilPerc),
            GpuUtilPerc = mean(gpuUtilPerc))
```

As the task.x.y data set contains no duplicates or missing values, there is no need for preparation of this data and we can begin integrating all three data files. To begin, we can combine the data in "gpu_summary" with "totalrender_data". Before joining these data sets we can verify that the "hostname" and "task_no" are primary and foreign keys.

```
totalrender_data %>%
  count(hostname,task_no) %>%
  filter(n > 1)
```

```
## # A tibble: 0 x 3
## # ... with 3 variables: hostname <chr>, task_no <int>, n <int>
```

```
gpu_summary %>%
  count(hostname,task_no) %>%
  filter(n > 1)
```

```
## # A tibble: 0 x 3
## # Groups:   hostname, task_no [0]
## # ... with 3 variables: hostname <chr>, task_no <dbl>, n <int>
```

```
sum(totalrender_data$hostname == gpu_summary$hostname) == length(gpu_summary$hostname)
```

```
## [1] TRUE
```

```
sum(totalrender_data$task_no == gpu_summary$task_no) == length(gpu_summary$hostname)
```

```
## [1] TRUE
```

So the hostnames and task numbers for both data sets are an exact match, and we can join them via these variables using the following code.

```
gpu_app_data = left_join(totalrender_data,gpu_summary, by = c("hostname","task_no")
```

Next, we can assign the metrics from "gpu_app_data" to each tile in the terapixel image by combining the data from "gpu_app_data" and "task.x.y". We can confirm that the variable "taskId" in "gpu_app_data" is both a primary key in this data set as well as a foreign key for "task.x.y".

```
gpu_app_data %>%
  count(taskId) %>%
  filter(n > 1)
```

```
## # A tibble: 0 x 2
## # ... with 2 variables: taskId <chr>, n <int>
```

```
task.x.y %>%
  count(gpu_app_data$taskId)
```

```
## # A tibble: 65,793 x 2
##    `gpu_app_data$taskId`                    n
##    <chr>                                <int>
##  1 00004e77-304c-4fbd-88a1-1346ef947567     1
##  2 0002afb5-d05e-4da9-bd53-7b6dc19ea6d4     1
##  3 0003c380-4db9-49fb-8e1c-6f8ae466ad85     1
##  4 000993b6-fc88-489d-a4ca-0a44fd800bd3     1
##  5 000b158b-0ba3-4dca-bf5b-1b3bd5c28207     1
##  6 000d1def-1478-40d3-a5e3-4f848daee474     1
##  7 000db9f9-d12d-4889-81cf-325906635535     1
##  8 0010651d-5f82-47ff-885c-1cdbaac2b1eb     1
##  9 00107991-1ad1-42c8-80b7-1c2dea75a1d5     1
## 10 0010aed5-8d4a-4298-9ff9-9a248f0db508     1
## # ... with 65,783 more rows
```

```
task.x.y %>%
  count(gpu_app_data$taskId) %>%
  filter(n>1)
```

```
## # A tibble: 0 x 2
## # ... with 2 variables: `gpu_app_data$taskId` <chr>, n <int>
```

We can therefore join both data sets simply using the "taskId" variable with the below code:

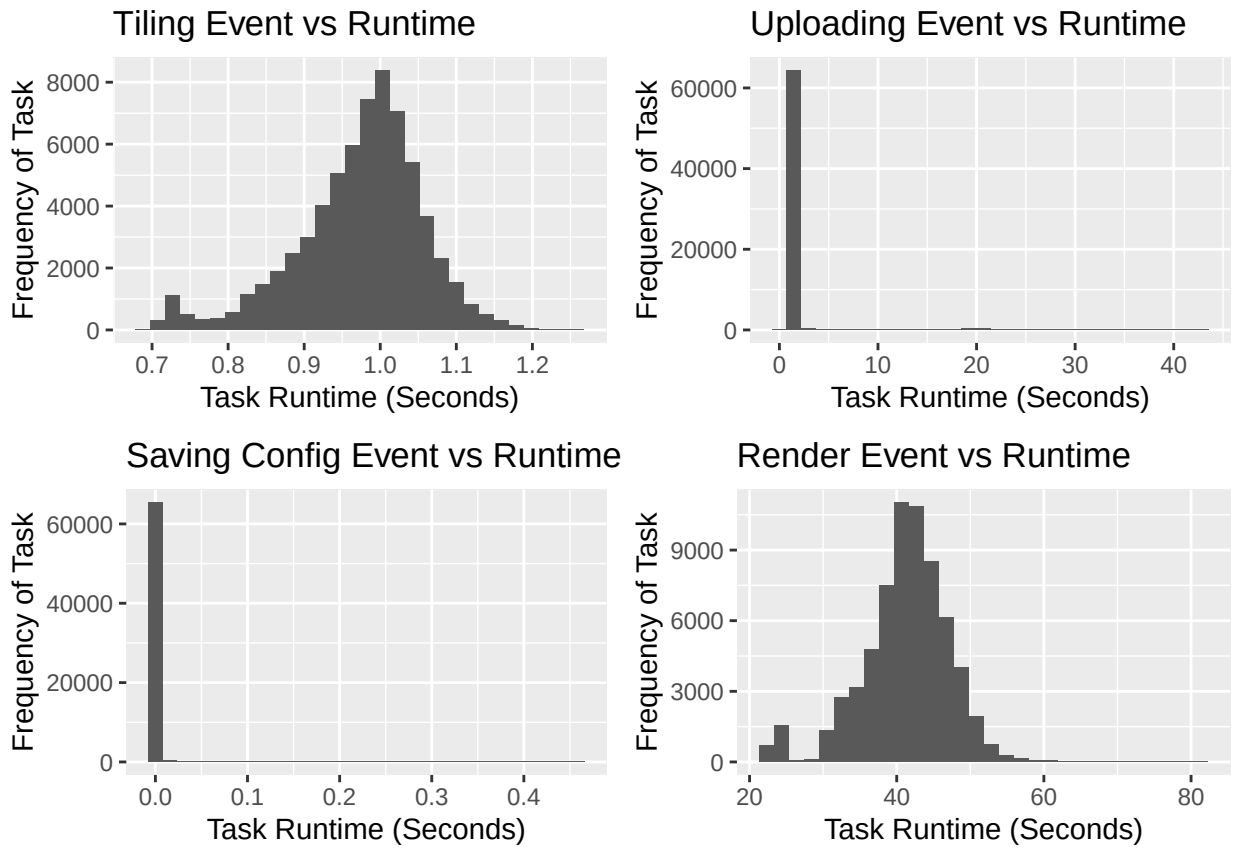
```
all_data = left_join(gpu_app_data,task_data[c("taskId","x","y","level")],by = c("taskId"))
```

With all data sets combined, we can begin analysis of the whole rendering process.

Modelling

Which event types dominate task runtimes?

Before comparing the runtimes of the different events in "app_wide", we should examine the distribution of the runtimes for each event individually to evaluate how best to compare them.



From the above plots we can observe some clear outliers for the "Uploading" event. Whilst the majority of the tasks have a runtime of around 1 second, there is a small peak at around 20 seconds. To gain a deeper understanding of the data, it would be helpful to investigate what is causing these disproportionately high runtimes. One possible avenue of investigation is to examine which virtual machines the high upload times occur on, to see if some virtual machines have issues executing the "Uploading" task. To check this, we can calculate the number of virtual machines that have higher than 10 second runtimes for the uploading task, take the unique values of these and then view the virtual machine hostnames with more than two occurrences.

```
length(app_wide[app_wide$eventName == "Uploading" & app_wide$runtime >10,]$hostname)
```

```
## [1] 1079
```

```
length(unique(app_wide[app_wide$eventName == "Uploading" & app_wide$runtime >10,]$hostname))
```

```
## [1] 793
```

```
app_wide %>%
  filter(eventName == "Uploading" & runtime > 10) %>%
  count(hostname) %>%
  filter(n>2)
```

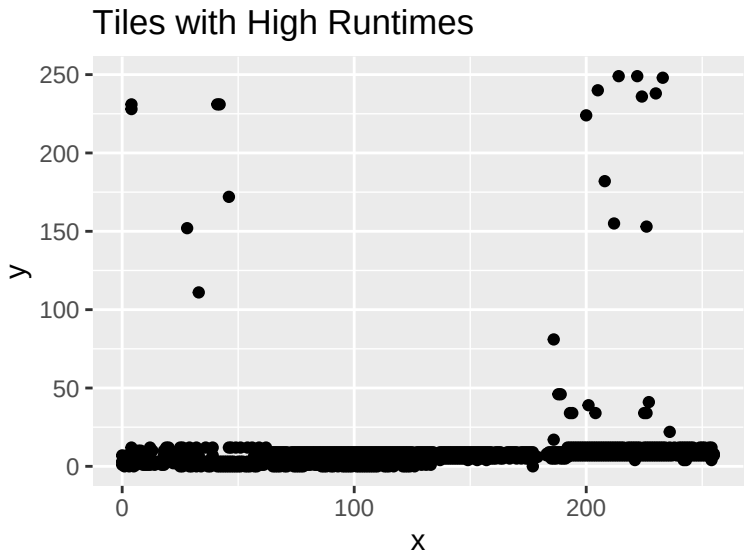
```
## # A tibble: 7 x 2
##   hostname                                  n
```

```
##   <chr>                                  <int>
## 1 0d56a730076643d585f77e00d2d8521a00000A     3
## 2 4a79b6d2616049edbf06c6aa58ab426a00000Q     3
## 3 4ad946d4435c42dabb5073531ea4f31500000M     3
## 4 8b6a0eebc87b4cb2b0539e81075191b900000E     3
## 5 8b6a0eebc87b4cb2b0539e81075191b900000L     3
## 6 95b4ae6d890e4c46986d91d7ac4bf082000001     3
## 7 dcc19f48bb3445a28338db3a8f002e9c000008     3
```

This indicates that there were 1079 observations in the data with eventName "Uploading" and a runtime longer than 10 seconds. Of these observations, they occurred on 793 different virtual machines where the maximum number of repeated occurrences on the same virtual machine was 3 - which happened on 7 different machines. With these outliers occurring on 77% of all virtual machines, it is unlikely that the high runtimes for the "Uploading" task are caused by a faulty or under-performing virtual machines. An additional route we can investigate is if there are particular attributes of the tiles that are being rendered that are conducive to a high upload time. We can display the tiles with high runtimes for the "Uploading" task with the following code. First we filter the app_wide" data to preserve only "Uploading" events with a runtime greater than 10 seconds, then we join this with "task.x.y" using the "taskId" variable as the primary key.

```
high_uploading_runtime = filter(app_wide,runtime>10 & eventName == "Uploading") %>%
  left_join(task.x.y,by = c("taskId"))
```

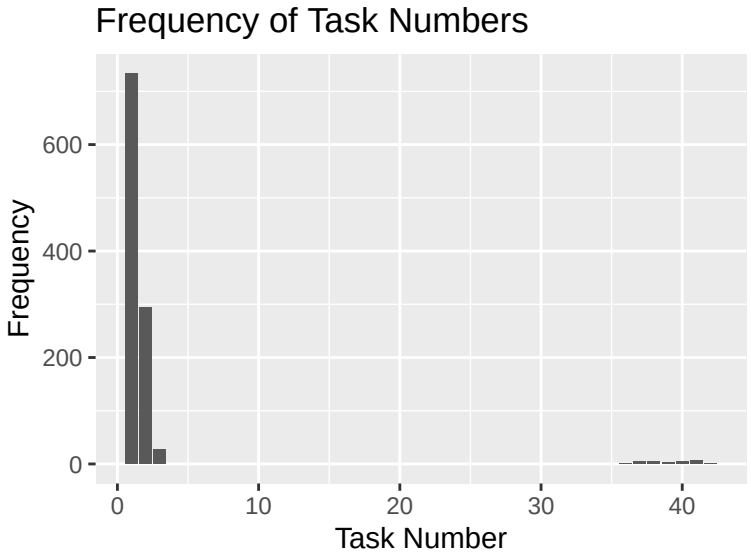
Now, we can visualise the tile locations with a high runtime for the "Uploading" task.



We calculate that 97.5% of these tiles are located where the y co-ordinate is less than 25. These are the tiles that are rendered at the beginning of the overall terapixel image rendering process. To evidence this, we can enhance "high_uploading_runtime" with the addition of the corresponding task number for each observation. Once again using the "taskId" as the primary key, we can assign each individual observation a task number using "all_data".

```
high_uploading_runtime = filter(app_wide,runtime>10 & eventName == "Uploading") %>%
  left_join(task.x.y,by = c("taskId")) %>%
  left_join(all_data[c("taskId","task_no")], by = c("taskId"))
```

Now, by plotting the distribution of the "task_no" we can see that it is heavily dominated by tasks 1 and 2. Indeed, tasks 1 and 2 constitute 95.18% of the total outlier data. From both the previous plot and the below, it is clear that the remaining outliers not in tasks 1 and 2 are varied both in terms of the task number and tile in which they occur. As these account for only 0.08% of the total tasks, they will not be investigated further in this analysis.



From this we can surmise that there are inefficiencies at the beginning of the terapixel rendering process that stem from the "Uploading" task runtime being significantly longer than it is during the remainder of the total process. To quantify this more precisely, after joining the "task_no" variable to "app_wide" from "totalrender_data" by using "taskId" as the primary key, we can compare the average runtime for the "Uploading" event of the first two tasks to the rest of the tasks.

```
#arithmetic mean of "uploading" runtime in first two tasks.
mean(filter(app_wide, task_no < 3 & eventName == "Uploading")$runtime)
```

```
## [1] 11.37057
```

```
#arithmetic mean of "uploading" runtime in tasks excluding first two.
mean(filter(app_wide, task_no > 2 & eventName == "Uploading")$runtime)
```

```
## [1] 1.073102
```

Therefore, the runtime for the "Uploading" event is on average 959.6% longer in the first two tasks than in the remaining. To determine exactly how many virtual machines this occurs on, we can find the number of virtual machines that have a runtime higher than the average "Uploading" runtime for all tasks excluding the first two.

```
runtime_average = mean(filter(app_wide, task_no > 2 & eventName == "Uploading")$runtime)

length(unique(filter(app_wide, task_no == 1 &
                       eventName == "Uploading" & runtime > runtime_average)$hostname))
```

```
## [1] 882
```

```
length(unique(filter(app_wide, task_no == 2 &
                       eventName == "Uploading" & runtime > runtime_average)$hostname))
```

```
## [1] 801
```

So in tasks 1 and 2, 882 and 801 virtual machines have longer than average runtimes for the "Uploading" event, corresponding to 86.13% and 78.22% of the total virtual machines in the cloud supercomputer.

We can perform similar calculations on the "TotalRender" task to assess how the total render time of each task is impacted by the high uploading times.

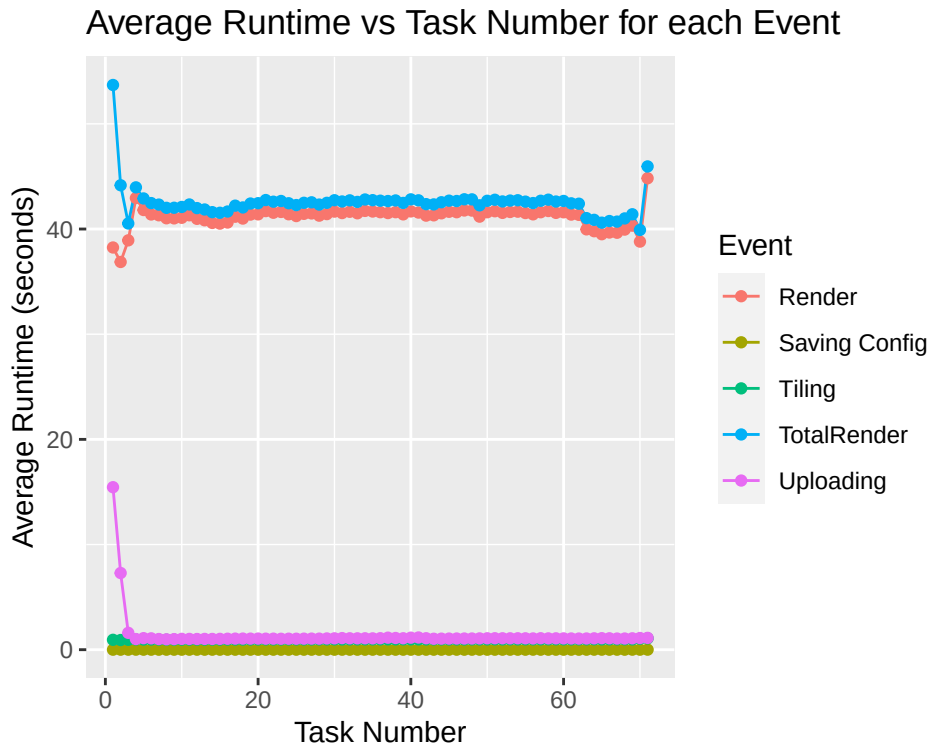
```
#arithmetic mean of runtime for "TotalRender" in first two tasks.
mean(filter(app_wide, task_no < 3 & eventName == "TotalRender")$runtime)
```

```
## [1] 48.92971
```

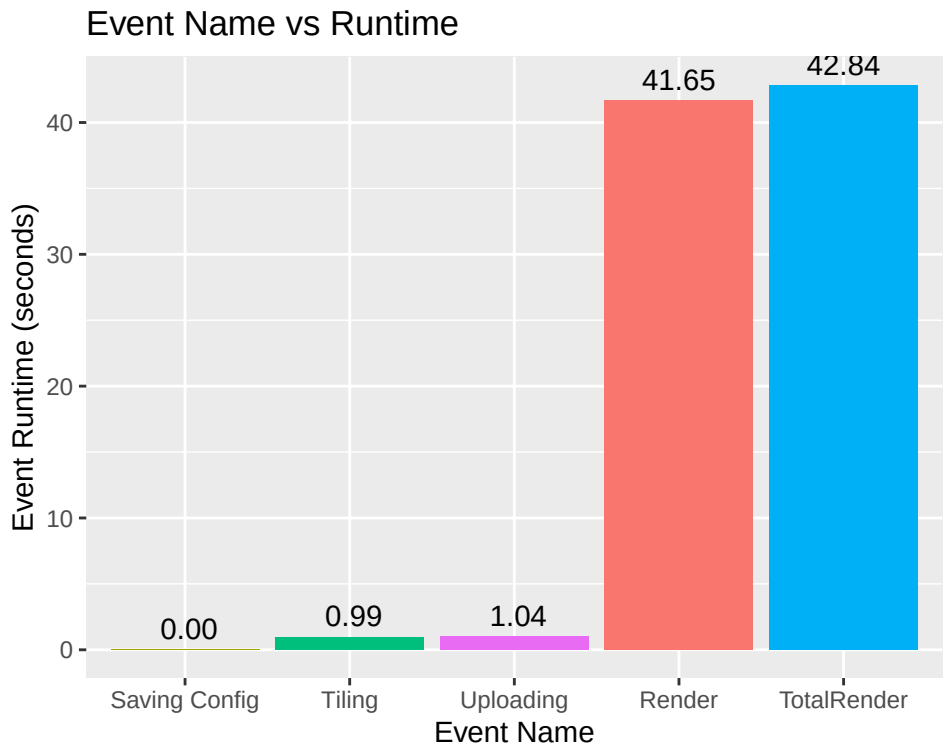
```
#arithmetic mean of runtime "TotalRender" in tasks excluding first two.
mean(filter(app_wide, task_no > 2 & eventName == "TotalRender")$runtime)
```

```
## [1] 42.40157
```

So the TotalRender variable is only 15.4% longer on average during the first two tasks compared to the rest. Whilst this increase is not huge, it is a point to consider both in the remainder of this analysis and for future implementation of this cloud architecture. A primary goal of this project is to ensure the scalability of the process, and whilst an increase of this magnitude when individual task runtimes are on average less than a minute is small, if these were substantially increased this inefficiency would be far more impactful. The results of the above analysis can be summarised in the following plot, showing the change in average runtime for each event and task.



This is an important plot to consider when comparing the runtimes for different events, as the arithmetic mean runtime for the "Uploading" event will be skewed due to the outliers primarily located in tasks 1 and 2. Due to the median being far more robust against outliers, it would be a more appropriate metric in comparing the central tendency of each event runtime. The median will also reflect a runtime more representative of the "Uploading" event in the majority of tasks. The below bar chart depicts the median task runtime for each event.



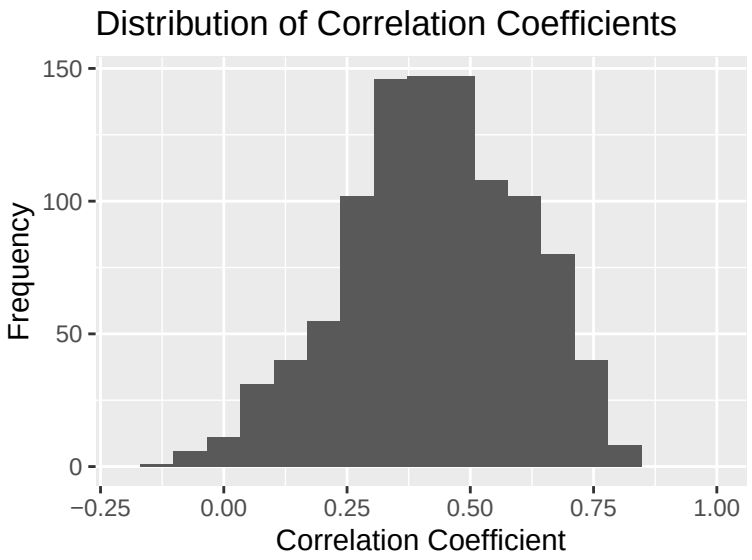
Clearly, "Render" dominates the overall process. This is followed by "Uploading", which is only 5.89% longer than "Tiling". Finally, "Saving Config" takes almost no time at all. An important point to consider with regards to the above plot, however, is that in the previous section we proved that the "Uploading" and "Tiling" events are initialised simultaneously. Therefore, only the event with the longer runtime will contribute to the overall total rendering time for each tile. By calculating the difference in render time for each "Uploading" and "Tiling" event per task, we can see the "Uploading" event is longer than "Tiling" every time. Therefore, the "Tiling" event does not contribute to the overall runtime of the total terapixel rendering process. The code that verifies this is given below:

```
runtime_diff = filter(app_wide,eventName %in% c("Uploading","Tiling")) %>%
  select(eventName,taskId,runtime) %>%
  pivot_wider(names_from = eventName,
              values_from = runtime) %>%
  mutate(diff = Uploading - Tiling)


sum(runtime_diff$diff > 0)/dim(runtime_diff)[1]
```

```
## [1] 1
```

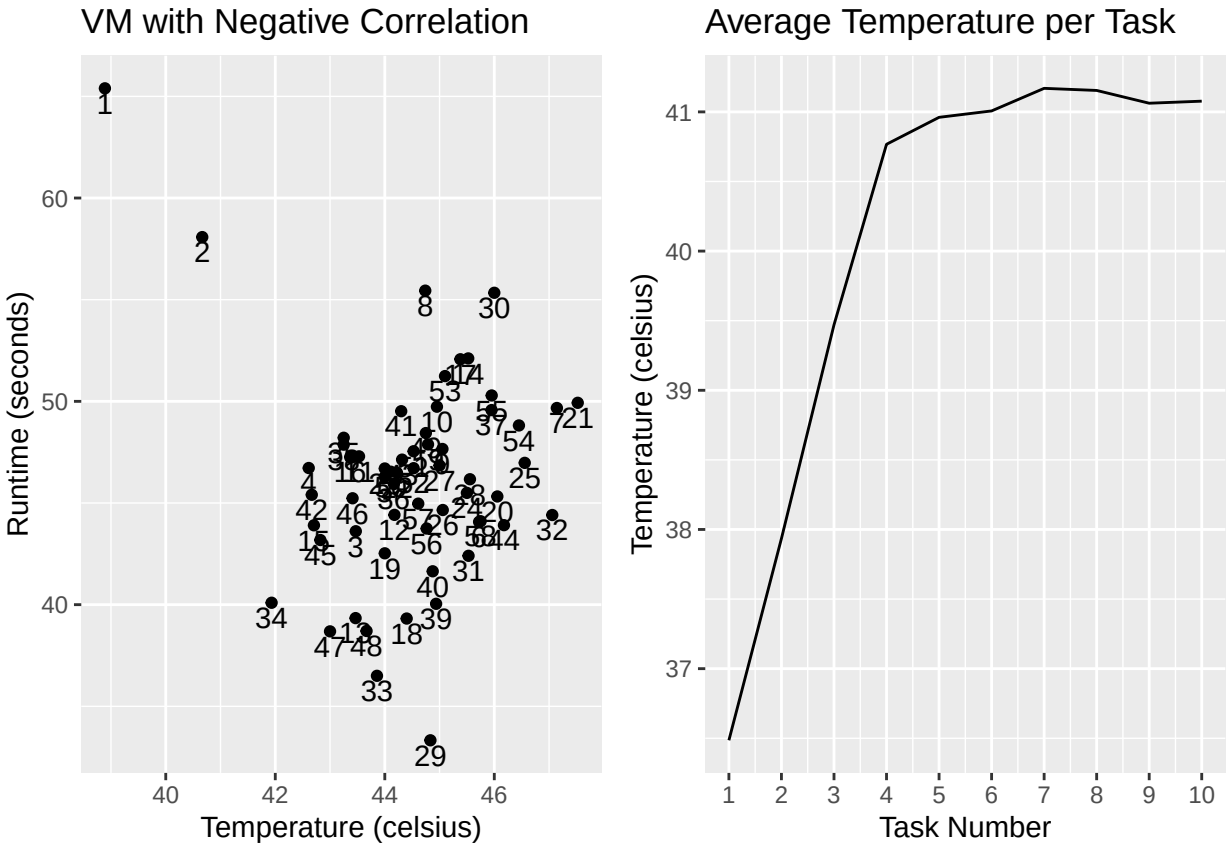
What is the interplay between GPU temperature and runtime performance?

As each virtual machine has a different GPU card, an analysis of the relationships between different variables should take this into consideration. For instance, a card that is perpetually cooler than most other cards, may still increase in temperature as task runtime increases, but will do so on a different scale incomparable to other GPU cards. Therefore, a comparison between temperature and runtime performance should only be made per virtual machine. To examine the correlation between temperature and runtime for each of the 1024 virtual machines used to render the terapixel image, we can calculate the correlation coefficient between both variables for each virtual machine, and visualise the distribution.



As the above histogram shows that these correlation coefficients are normally distributed, we may apply parametric techniques in the analysis of them. Moreover, due to each correlation coefficient being calculated from samples of various size, different weights will need to be assigned to them within these calculations. The weighted arithmetic mean of the above distribution is 0.43 with a weighted standard deviation of 0.18. Whilst these indicate a mostly positive correlation between these variables, the weighted standard deviation shows a large degree of spread and this is echoed by the above distribution, with correlation coefficient values ranging from -0.13 to 0.82. The above histogram also supports our approach of assessing this relationship per virtual machine, as the correlation coefficient taken of both variables over the whole data set is 0.13, which is clearly not representative of the distribution shown above.

To more closely examine the causes of this variance, we can create at a scatter plot of temperature vs runtime, with the task number labeled, for the virtual machine with a negative correlation of -0.13. This will be supplemented by a plot showing the average temperature for the first 10 tasks.

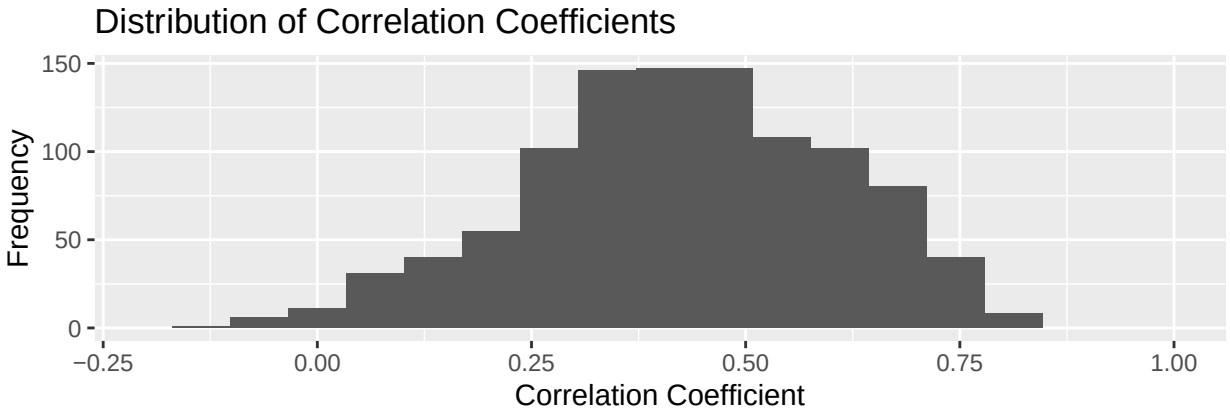


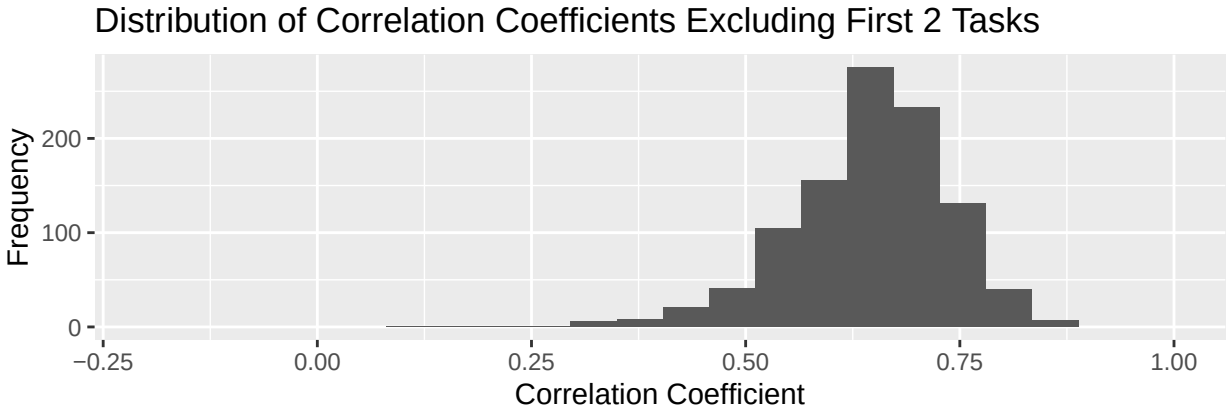
The above plots indicate that the negative correlation between temperature and runtime for this virtual machine is driven by the first two tasks in the rendering process. As discussed in the previous section of this report, the first two tasks in the total rendering process have disproportionately high uploading times, which consequently gives many of the virtual machines longer than normal total rendering runtimes. This is then compounded by a lower than average temperature for the first 3 tasks in the total rendering process that can be seen for all GPU cards, presumably as they heat up. For the virtual machine above, this gives the impression that as the runtime decreases, the temperature increases, but this is a special case for the first two tasks and not representative of the remainder of the rendering process. To evidence this, we can recalculate the correlation coefficient for the virtual machine, while excluding the observations for the first two tasks. This is performed in the below code using custom function **calculate_corr()**.

```
calculate_corr(unique_hostnames[match(min(temp_vs_runtime_corr),
                                      temp_vs_runtime_corr)],
               "runtime","tempC",task_number = 2)
```

```
##             tempC
## runtime 0.3463126
```

Therefore, with the removal of of the first two tasks in the data, the correlation coefficient for this virtual machine has increased dramatically from -0.13 t0 0.34. To see if this result is repeated for all virtual machines, We can recalculate the correlation coefficients for all virtual machines in the data without the first two tasks, and compare the correlation distributions with the inclusion and exclusion of the first two tasks for all virtual machines.

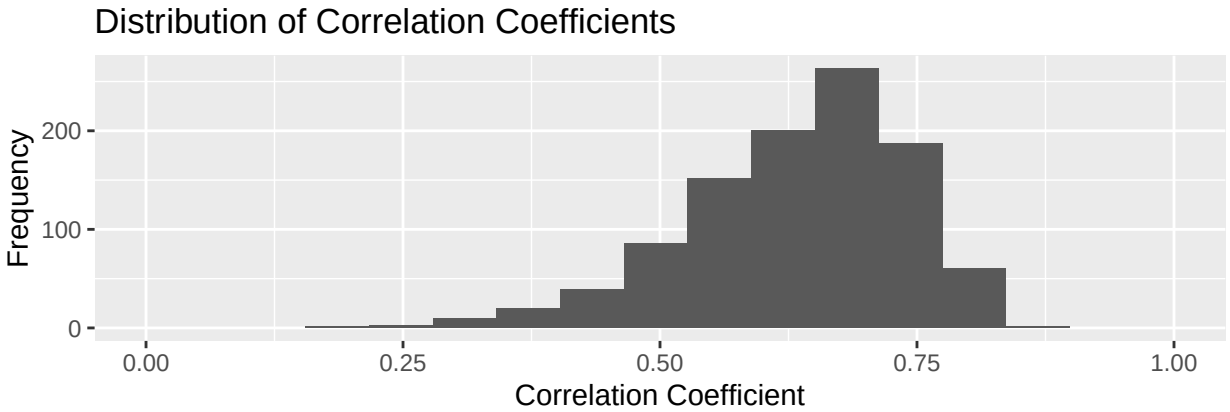


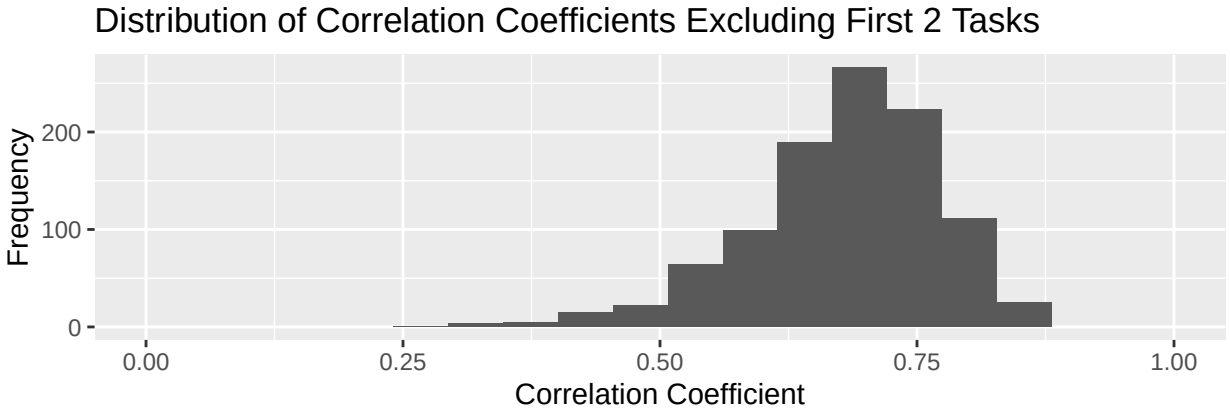


Clearly, the correlation coefficients calculated when omitting observations from the first two tasks are significantly more positive. The weighted mean of this distribution is 0.64 with a weighted standard deviation of 0.09. This shows the correlation coefficients are far more positive with much less variance, indicating that the first two tasks had a substantial effect in negatively skewing the correlation coefficients. As the first two tasks only constitute 3.11% of the average total amount of tasks performed on each virtual machine, we can say for the vast majority of the time there is a moderate positive correlation between temperature and runtime. This suggests that as the runtime of a task increases, so will the temperature of the GPU, and vice versa.

What is the interplay between increased power draw and render time?

We can approach this analysis in the same way as in the previous section, by calculating the correlation coefficient for the power draw and render time for each virtual machine. This method takes into account that each virtual machine represents a unique environment and may have different power consumption properties. As before, we can view the distribution of correlation coefficients for these variables in a histogram. Additionally, we can recalculate the correlation coefficients when exempting observations from the first two tasks to assess if these negatively skew the results as they did in the previous section.



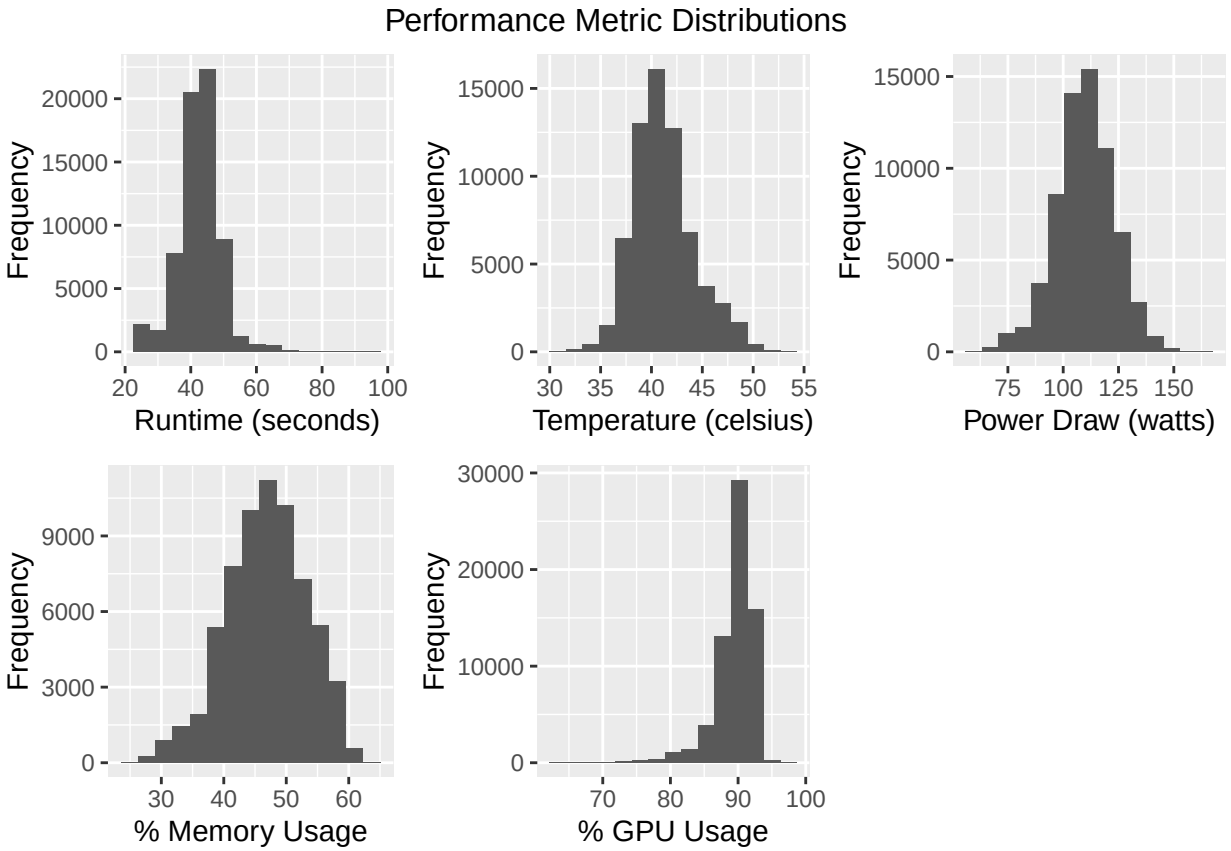


The above plots indicate that for all virtual machines in both scenarios, there exists a positive correlation between the runtime and the power draw of the GPU. Compared to our previous analysis of the interplay between temperature and runtime, the first two tasks have a limited impact in skewing the results. As both distributions are normal, we may use parametric techniques to support this assertion with quantitative assessments. The weighted mean and standard deviation of the correlation distribution with all observations are 0.63 and 0.11, respectively. Likewise, for the correlation distribution where the first two tasks are excluded, the weighted mean and standard deviation are 0.68 and 0.09, respectively. We can therefore see that observations from the first two tasks do negatively skew the distribution and increase the variance, however in this case the magnitude of this effect is minimal. In both scenarios, however, we can observe a moderate positive correlation between power draw and render time, suggesting that as one of these variables increases, so will the other.

Another interesting facet of the interplay between these variables, is that it seems far more consistent over all virtual machines. When calculating the correlation coefficient using all of the data, rather than per virtual machine. for variables temperature and runtime, we observed a far reduced correlation coefficient than the weighted mean average of the correlation coefficients per virtual machine. This was reflective of the large range of different temperatures and runtimes across various virtual machines. In the case of variables power draw and runtime, the correlation coefficient calculated for these variables is 0.58. This value is far more representative of the above distributions and indicates that the values involved in this interplay are more consistent across different virtual machines than the temperature values.

Can we quantify the variation in computation requirements for particular tiles?

Before quantifying the variation in the computation requirements for different tiles, we should visualise the distribution for each performance metric to determine which techniques can be implemented in our calculations.

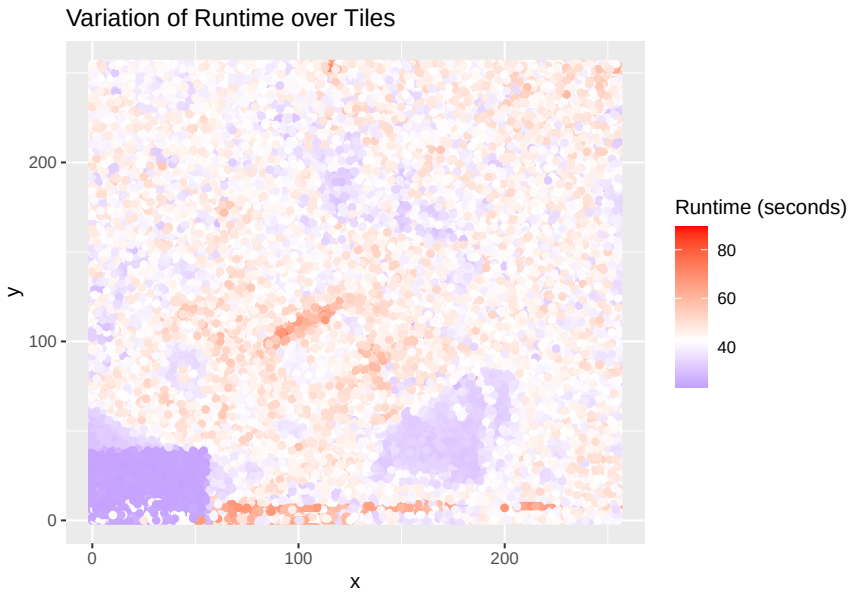


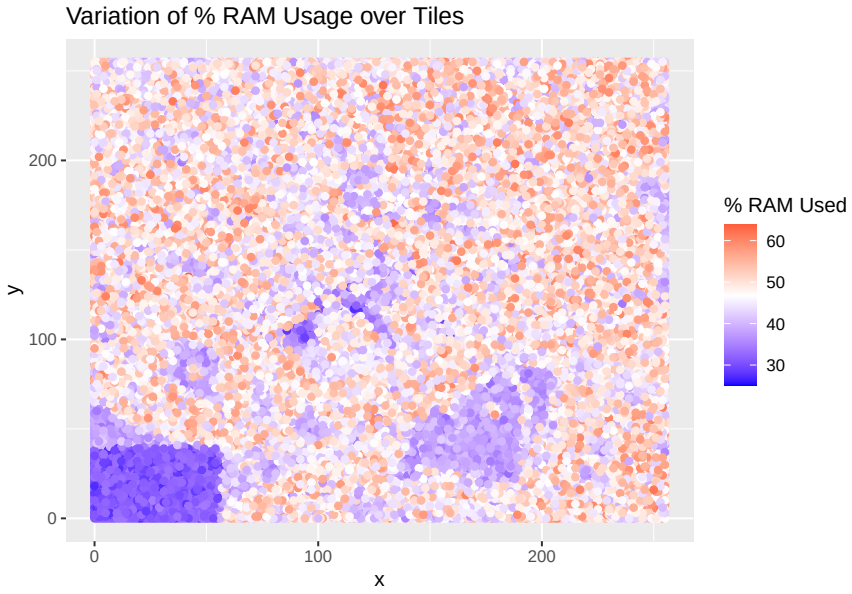
Clearly, each variable is normally distributed with no obvious outliers, so we may use parametric techniques to quantify their variation. The below table details the mean, standard deviation and coefficient of variation for each variable.

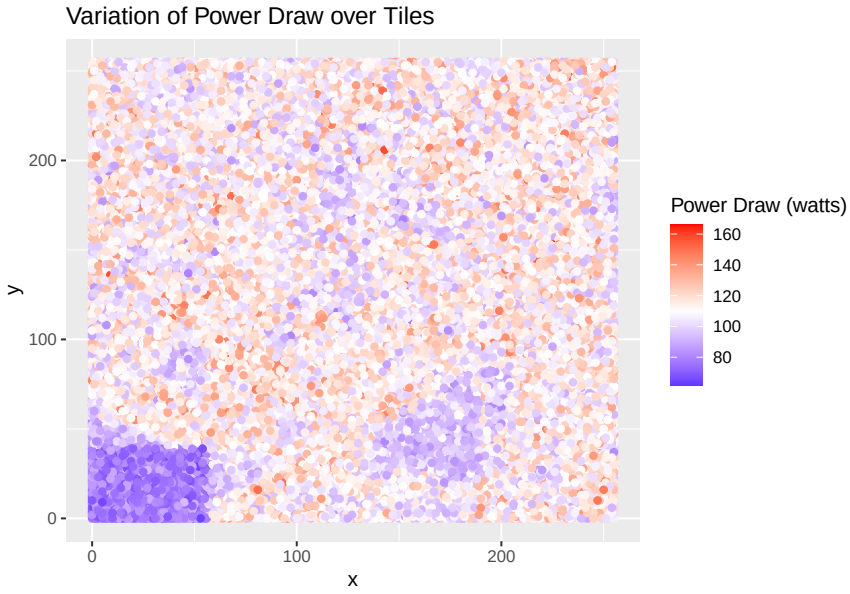
Metric	Mean	Standard Deviation	Coefficient of Variation
Render Time (seconds)	42.60	6.50	0.15
Power Draw (watts)	109.83	13.27	0.12
Temperature (celsius)	41.20	2.99	0.07
% Memory Usage	46.77	6.40	0.14
% GPU Usage	89.49	2.93	0.03

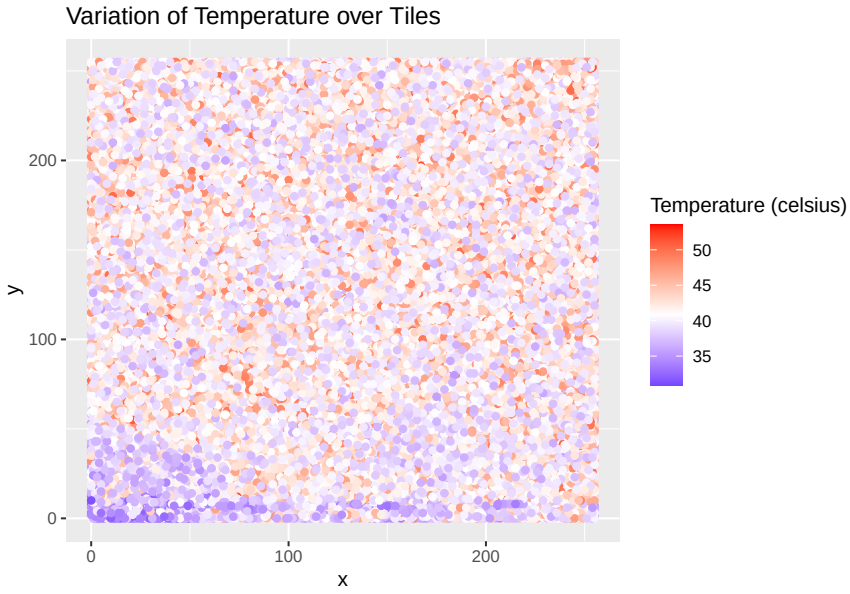
Using the coefficient of variation as the basis to compare different variable variances, it is evident that the variables "runtime", "powerDraw" and "MemUtilPerc" vary the most across different tiles. Having said this, objectively the variation across the tiles is limited, with the maximum coefficient of variation being 0.15.

To gain a more intuitive understanding of which particular tiles display the most variation in computation requirements, we can plot the x and y co-ordinates of each tile, and use colour to highlight the variation of a chosen variable. The below plots depict the variation of runtime, % Memory Usage and Power Draw across the tiles in the terapixel image.


Variation of Runtime over Tiles
200
100
0
y
0
100
200
x
Runtime (seconds)
80
60
40

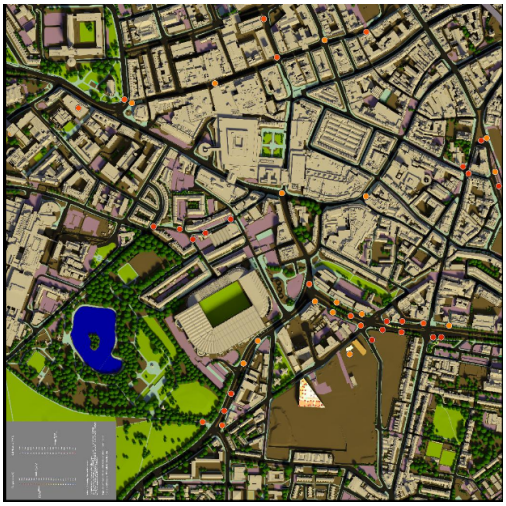

Variation of % RAM Usage over Tiles
200
100
0
y
0
100
200
x
% RAM Used
60
50
40
30



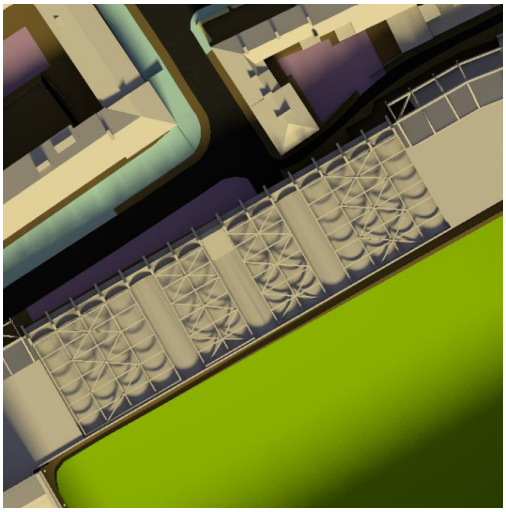


In the above plots a divergent colour scale is used to highlight particular areas that are substantially above and below the average value for that variable. A clear trend in all three plots is that the areas with less computational requirements are consistent. For the plots of runtime and power draw, this supports our earlier conclusion of a positive correlation between both variables, and provides evidence that the same is true with these variables and % Memory Usage. In contrast, the plot showing the variation of temperature across the tiles shows no clear trend. This is indicative of the different GPU environments, and how the different cores operate at different working temperatures. The only clear trend, is the tiles that are first rendered at the bottom of the plot which are cooler than the rest, as proved in a previous section.

As the above plots render the terapixel image rotated anti-clockwise by 90 degrees, we will perform the same transformation to the final terapixel image and cross-reference them.



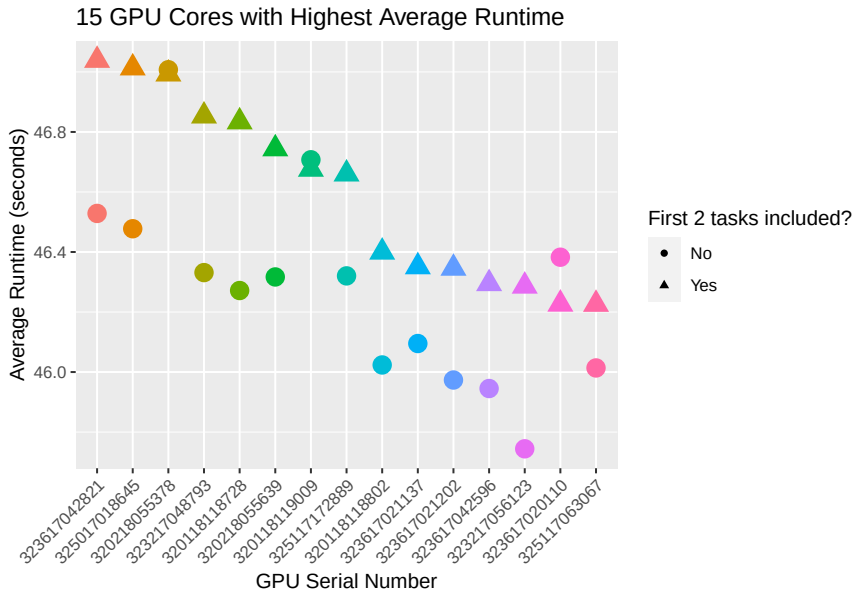
When comparing the final image with the plots depicting performance metric variation, we can observe lower computational requirements for the large areas of the image of the same colour. In contrast, we can observe much higher runtimes for areas where render tracing has been used extensively to depict shadows. A particular example of this in the above runtime plot is at around (100,100), which renders the part of the terapixel image shown below.



Can we identify particular GPU cards (based on their serial numbers) whose performance differs to other cards? (i.e. perpetually slow cards).

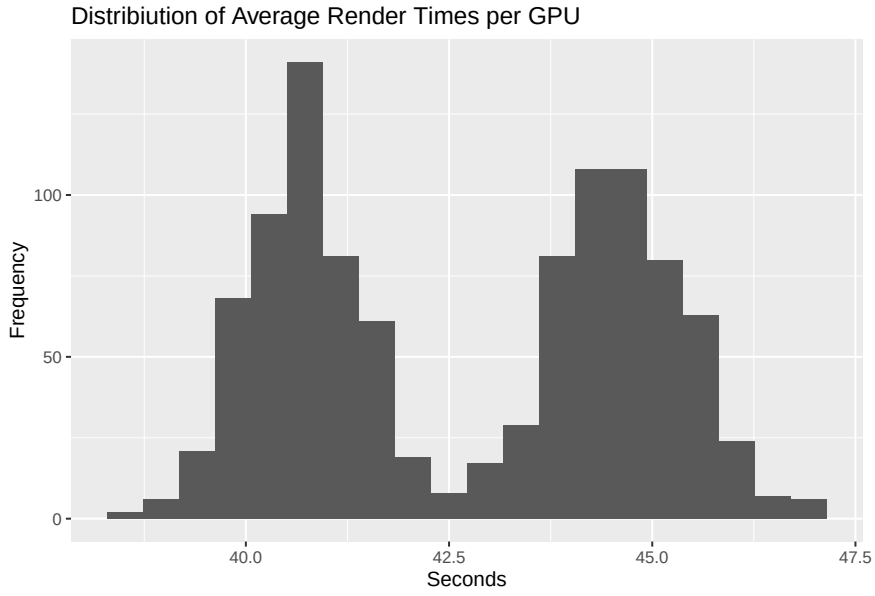
To identify GPU cards with distinct performance properties, we can summarise "all_data" by taking averages of each performance metric per virtual machine. However. it is worth recalling that the first two tasks performed on the majority of virtual machines exhibit behavior that is not representative of the GPU card performance over the remainder of the total rendering process. Taking this into consideration, we can create two data summaries from "all_data" providing the average performance metrics for each virtual machine. One data set will contain data from all tasks, whilst the second will exclude observations made in the first two tasks performed on each virtual machine. Using both of these data sets, we can identify GPU cards that express extreme properties and verify that this is the case for all tasks, and are not being heavily skewed by abnormalities in the first two.

The below plot depicts 15 different GPU cards with the highest average render time per tile, in descending order by the average including observations from the first two tasks. It shows two averages, one calculated including the first two task observations and one without.



For the majority of the GPU cards, the average total render time per tile increases, on average, by 0.2 seconds with the inclusion of the first two tasks. Therefore, these observations do skew the averages slightly but not enough to cause a significant alteration in the data. Additionally, while the inclusion slightly alters the order of the 15 GPU cards with the highest average total render time per tile, these cards remain the slowest of the 1024 GPU cards.

We can also observe the distribution of the average total render time per tile for each GPU, calculated including the observations for the first two tasks. It is clear in the below plot that there are two main groups of GPU cards that differ in runtime performance.

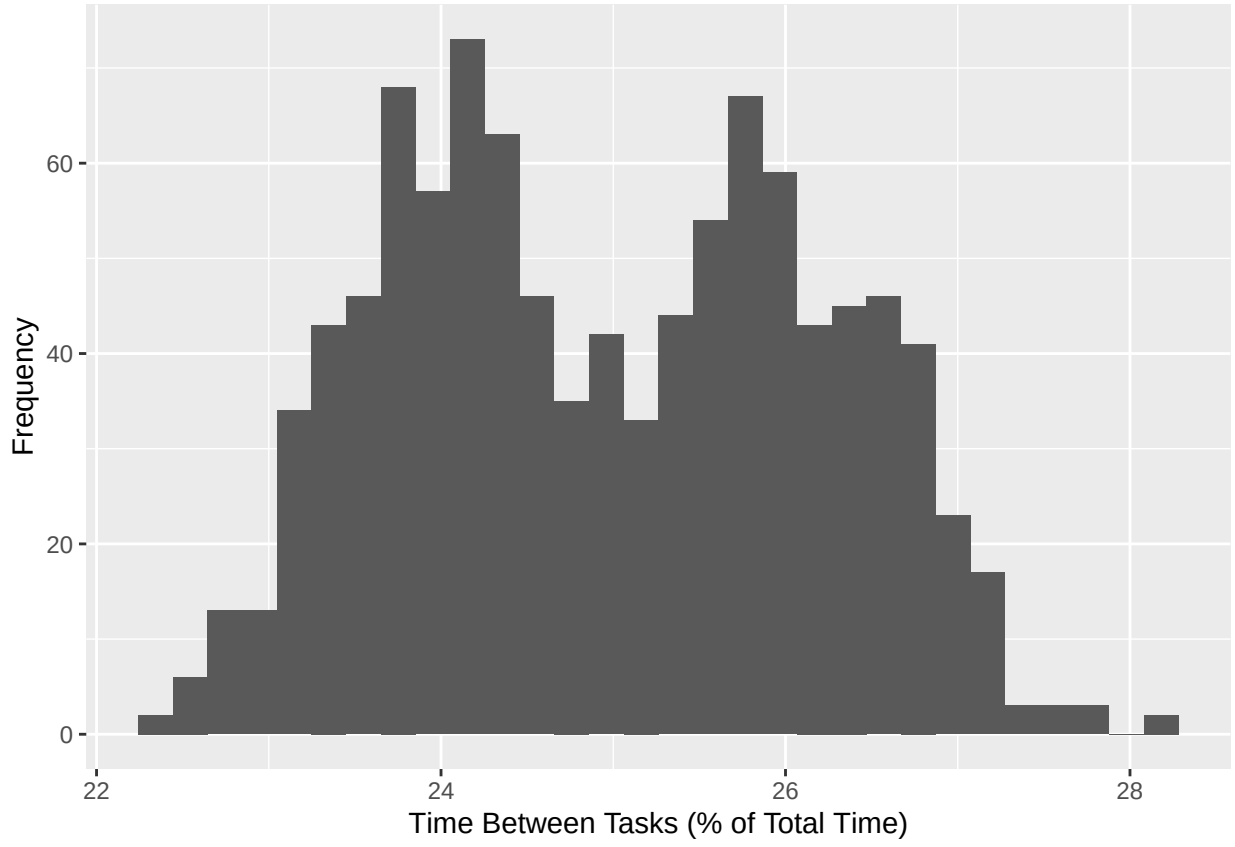


What can we learn about the efficiency of the task scheduling process?

From our analysis of the "gpu" data, we found that this data set contains snapshots of each virtual machine every two seconds. This revealed periods between tasks where the GPU card was not being utilised. Whilst we used this previously to assign task numbers to each virtual machine tile rendering process, we can also use it to establish how efficient the task scheduling process is. The more efficiently the tasks are scheduled, the less time each virtual machine will spend not running in between tasks. Therefore, we can investigate this by finding the proportion of observations where the gpuMemUtil and gpuUtilPerc variables are equal to 0 (indicating that a task is not running) from the beginning of the first task to the end of the final one. To do this, we can use the below function:

```
waiting_time_perc = function(hostn) {
  filt_data = filter(gpu_data,hostname == hostn & task_no != 0)
  last_task_ref = min(which(filt_data$task_no == max(filt_data$task_no) & filt_data$gpuMemUtilPerc == 0
  filt_data = slice(filt_data,1:last_task_ref)
  perc_zero = sum(filt_data$gpuMemUtilPerc == 0 & filt_data$gpuUtilPerc == 0)/dim(filt_data)[1] *100
  return(perc_zero)
}
```

This function returns the percentage of observations where the gpuMemUtil and gpuUtilPerc variables are equal to 0 for a given virtual machine hostname. It does not count any observations taken before the beginning of the first task or after the final task has finished. We can apply this function to all 1024 virtual machines and visualise the distribution of the result.



In addition to the distribution, the minimum and maximum percentages of time between tasks for a virtual machine are 22.3 and 28.2, respectively. The mean average percentage is 25. So on average, approximately

a quarter of the total rendering time for the whole terapixel image is spent by a virtual machine waiting to be assigned the next task.

The above plot resembles the histogram in the previous section showing the distribution of the average total render time per tile for each virtual machine. We can create a small data set to investigate whether average runtime correlates with average time spent waiting for a task to be assigned to the virtual machine by joining up the data and computing the correlation coefficient with the below code.

```
vm_wait_data = as.data.frame(cbind("hostname" = unique_hostnames,vm_waiting_times)) %>%
  left_join(filter(slow_cards,runtime_env == "runtime_exc"),by = c("hostname"))

cor(as.numeric(vm_wait_data$vm_waiting_times),as.numeric(vm_wait_data$avg_runtime))
```

```
## [1] -0.8698723
```

This indicates that there is a strong negative correlation between the average task runtime and the percentage of the total rendering process spent waiting for a task to be assigned for each virtual machine. Therefore, as the average runtime for a task increases, the time spent waiting for a task to be assigned decreases.

Evaluation

Evaluate Results

Initially set out at the beginning of this report, our goal was to rigorously evaluate cloud supercomputing as a method to create terapixel images. To achieve this goal, several objectives for the analysis were given. Upon reflection, it is clear that these objectives have been met. Each objective has been clearly and methodically approached and given a corresponding subsection within the "Modelling" section of this report. As part of this critical evaluation, we made some key findings.

First of all, it was established that the "Tiling" and "Uploading" events began concurrently. We then showed that the "Uploading" event is always longer than the "Tiling" event, therefore "Tiling" makes no contribution to the overall rendering time for each task. Another important finding was that at the beginning of the rendering process, primarily in the first task on 86% of the virtual machines and second task on 78%, the "Uploading" event takes significantly longer than during the remainder of the full rendering process. Specifically, on the average runtime over these two tasks is 10.6 times longer than the average over the remainder of the process for the "Uploading" event. This observation meant that the total rendering times for these tasks were significantly skewed, and our further analysis of the data would take this into account.

We also showed that there is a positive correlation between both total render time and temperature, as well total render time and power draw, This analysis was performed by calculating the correlation coefficients for these variables on each virtual machine to account for any differences in the GPU card. The differences in these GPU cards was investigated further in our later analysis, where the 15 slowest GPU cards were determined from the data by finding the highest average total render times per tile.

The variation of the performance metrics across the different tiles in the image was assessed using parametric techniques such as the arithmetic mean and standard deviation. These were used once the normality of the variable distributions was verified. Finally, we analysed the task scheduling process and found that on average 25% of the total render time was spent by virtual machines waiting for a task to be assigned. This result means there is certainly advancement to be made in terms of task scheduling for cloud supercomputing processes.

Review Process

Due to the large size of the data sets, it was important to ensure that any processes ran on the entirety of it were well optimised. To this end, object growth has been avoided throughout the analysis, meaning

that should this analysis be performed in even larger data sets we can expect to code to scale well. To further promote efficiency, functions such as “apply” and “lapply” were used heavily throughout the analysis in place of “for” loops that lead to inefficiency. One of the few instances of a “for” loop being used in the preprocessing section is within the “assign_task_no” function. To avoid object growth, a vector of a specific numeric length was created prior to the for loop being ran in order to avoid incremental object growth.

As well as being well optimised, the data pipeline created for the analysis in this report is fully reproducible. By making use of Project Template, the analysis can be replicated using only the source and data files. Once the ProjectTemplate library is loaded and the working directory for R is set to the project location, running the “load,project()” function will load the data and perform the munging operations that preprocess the data into. The plots in the analysis that are based off this will then be automatically updated and this will be fed through into the final interactive shiny application. The only requirements are that the data is formatted the same as the data files used in this iteration of the project analysis.

An area of the analysis that could be improved would be to make more use of the granular data obtained from “gpu”. For our evaluation, an arithmetic mean has been used to describe the performance metrics for each task per virtual machine. To investigate the behaviour of these performance metrics in more detail, the data at the most granular level could have been used and may have revealed further insights about the rendering process and the GPU cores. This is a clear area that could be built upon if further analysis of the terapixel rendering process was to be undertaken.

Deployment

The key findings from the analysis detailed in this report are deployed in an interactive shiny app. This app is based on the data in this report, thus ensuring it’s reproducibility and ease to update should the analysis be performed on a new data set. It allows the user to look at the interaction between different performance metrics over different virtual machines, whilst providing a correlation coefficient for comparison with some of the analysis detailed above. Other key findings shown in the app include the variability of the performance metrics for each tile cross-referenced with the final image, as well as the event timelines. ELABORATE