Terapixel Project

Michael Luke Battle

30/12/2020

Project Requirements

Written Report Outline

What is the need for the project?

Justify your choice of response (i.e. the nature of, and your plan for, your project). To give strength to your argument you should reference to practice elsewhere (e.g. in academic literature, or industry practices)

Discuss and implementation of CRISP-DM

How successful has it been? Provide evidence, using appropriate evaluation methodologies, and comment on the strengths/weaknesses of your evidence in answering this question

What are the future implications for work in this area? If applicable, which areas of extension work are now possible due to the foundational work you have performed in this project?

A brief reflection on your personal and professional learning in undertaking this project. Here, you can comment on how you found the process, what you learned about the technologies and methodologies you used, which aspects you found most difficult/straightforward, and any conclusions which will inform the way you undertake similar projects in the future

Structured Abstract

REMEMBER TO INCLUDE STRUCTURED ABSTRACT

EDA Findings.

You should also produce additional documentation detailing the findings from your exploratory analysis. You are encouraged to make use of a literate programming framework, e.g. R Markdown, to align analytic code with narrative tetx. You should submit the source file(s) for the notebooks as well as output saved in PDF Format. No limit on length for this document!

CRISP-DM Methodology:

Business Understanding

Determine Business Objectives

Background

Maybe include more background information on Newcastle University and data

With world-leading research in data, Newcastle University is always undertaking new challenges to advance the field. In order to keep up with ever-expanding amount of data produced by cities, one such challenge is to be able to effectively convey the information from this data to stakeholders. To address this issue, the University has created a terapixel image of the city of Newcastle upon Tyne, including environmental data such as temperature and humidity, from sensors across the city. This terapixel image contains over one trillion pixels and not only allows the entire city to be viewed, but is rendered at 12 different levels allowing users to zoom in whilst retaining full picture detail. 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. This facilitates a far quicker rendering process, which is more suitable for a primary requirement of the project - to ensure the terapixel image supports daily updates.

Business Objectives

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.

Business Success Criteria

Assess Situation

Determine Data Mining Goals

Produce Project Plan

Data Understanding

Describe Data

In order to evaluate the rendering process, Newcastle University has provided three files detailing various aspects of a single run of the rendering process. The file "application.checkpoints.csv" has 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" "0d56a730076643d585f77e00d2d85
##  $ 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 the "eventType" can be either "START" or "STOP" to indicate the end of beginning of an event, and "eventName" can take the following values;

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

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

In which "TotalRender" is a combination of all other events, which represent different parts of the whole rendering process. In addition to "eventType" and "eventName", "timestamp" gives the time associated with this event starting or stopping, "hostname" is the hostname of the virtual machine auto-assigned by the Azure batch system and "jobId" and "taskId" are the unique IDs of the Azure batch job and task, respectively.

The file "gpu.csv" has the following dimensions and variable names, with all variable values being of class character:

```
dim(gpu)
```

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

```
colnames(gpu)
```

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

```
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 variable names "timestamp" and "hostname" have the same meaning as in "application.checkpoints.csv". "gpuSerial" and "gpuUUID" are the serial number and unique ID of the physical GPU card, which represent a one-to-one match the "hostname" variable. As there are 1024 gpu cores, 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
```

The additional variables in this data set are quantitative variables describing different aspects of the 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.csv". 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.csv", whilst "x" and "y" represent the location of each pixel 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 and rendered whilst the other levels are derived in the tiling process. Level 12 is when the the image at maximum zoom, and level 1 is when the image is zoomed right out. The "jobId" relates to the "Level" variable, so in total there are three different "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, it is necessary to assess the quality of it. First 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
```

As we are satisfied that 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
```

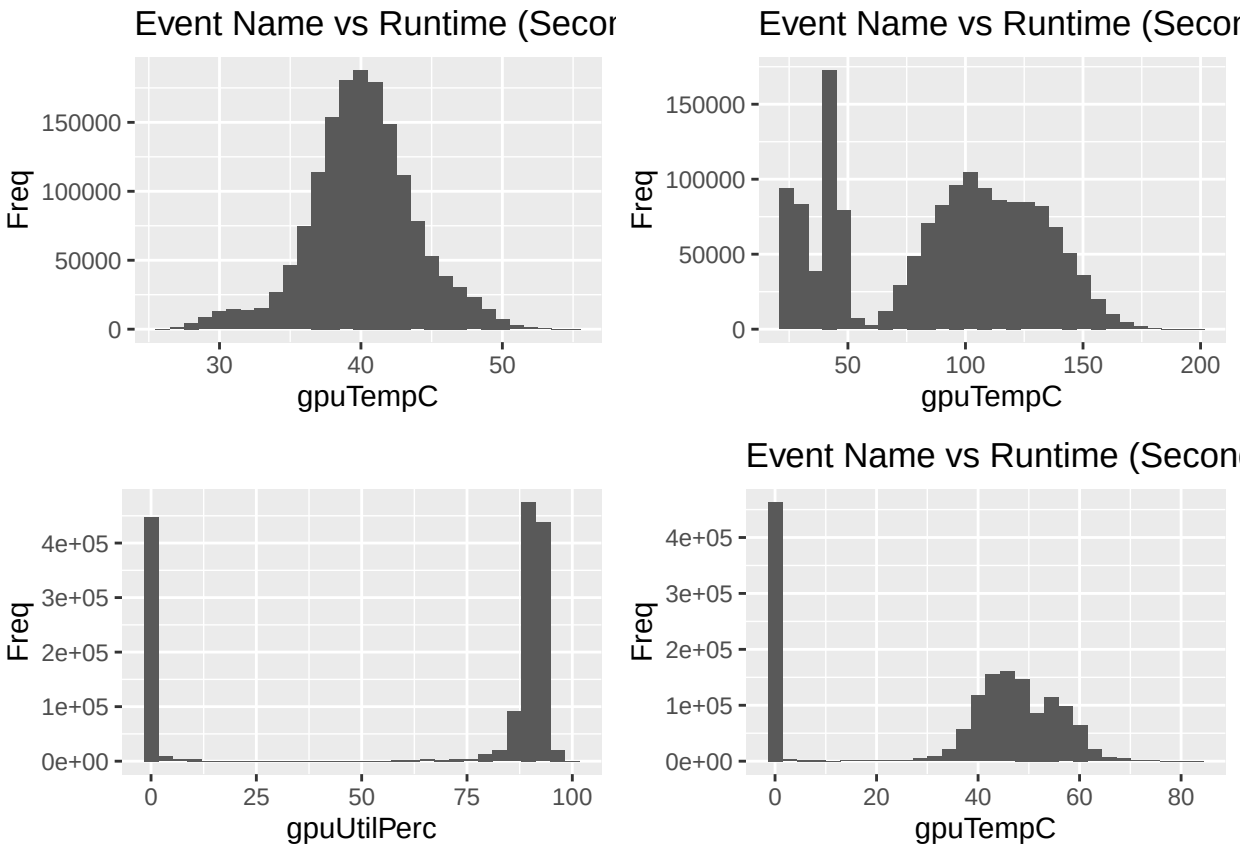
```
dim(task.x.y)[1] - dim(unique(task.x.y))[1]
```

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

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

Although there is no missing data in the sense of NULL values, "gpu" has a significant population of outliers. We can observe this by plotting histograms for each quantitative variable in the data set.

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



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

Data Preparation

In order to rigorously evaluate the supercomputer rendering process, 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".

As the data in "application.checkpoints" is clean, we can proceed with using it to transform the data into a more useful format whilst deriving new attributes from it. To make the "timestamp" variable more meaningful, we can convert it from class character to datetime format, or POSIXct. As each value in

"timestamp" is of the form "2018-11-08T07:41:55.921Z", we can use the below function to remove the letters "T" and "Z".

```
clean_date_time = function(string) {
  string_rep = str_replace(string, "T"," ")
  string_rep2 = str_replace(string_rep, "Z","")
  return(string_rep2)
}
```

Then, the resulting "timestamp" values are parsed into into POSIXct form, and the duplicates that were found in the *Data Understanding* section of this report are removed.

```
#apply clean_date_time function to date_time in data and create list
date_time_chr = lapply(application.checkpoints$timestamp,clean_date_time)

#create list of date-times converted to datetime format
date_time = parse_date_time(date_time_chr,"%Y%m%d %H%M%S")

#change timestamp to converted date_time format
app_data$timestamp = date_time

#remove duplicates
app_data = unique(app_data)
```

With the "timestamp" variable in a more computational-friendly format, we can now derive the runtime for each event. First, the data must be converted from long format to wide format with respect to the eventType variable. Then, the start time of the event can be subtracted from the stop time to calculate the runtime. 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)

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

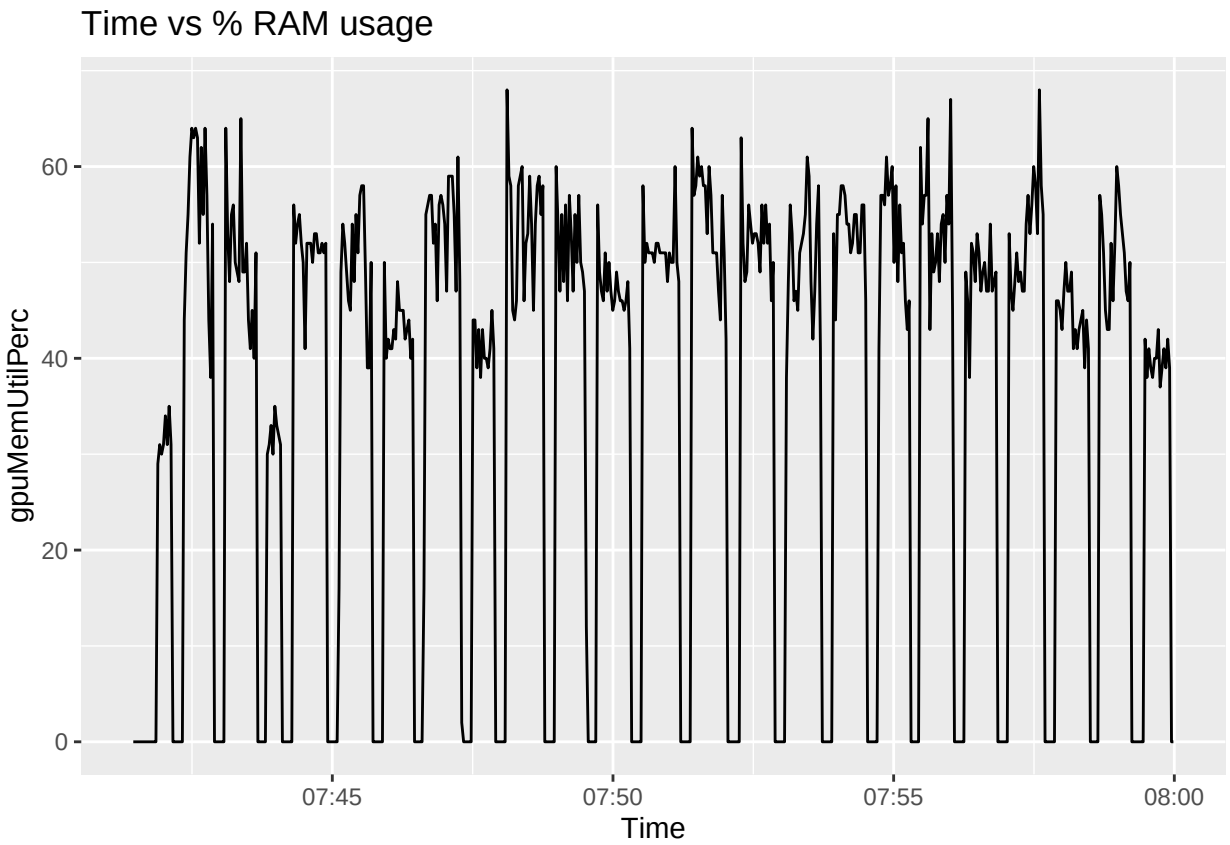
In order to view the interaction of the "runtime" variable with variables from other data sets, it would be useful to create a new data set that exclusively contains eventName "TotalRender", thus showing only the total runtime for each rendering task per virtual machine. To further enhance this data set and make it conducive to joining with the "gpu" data later in the analysis, a "task_no" variable will also be added, reflecting the chronological order that each rendering task is performed per virtual machine. These steps are performed using the following code:

```
totalrender_data = filter(app_wide,eventName == "TotalRender") %>%
  arrange(hostname,START)

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
```

In the above code, the totalrender_data is created by filtering app_wide where eventName is equal to "TotalRender", and then the data is arranged by the virtual machine hostname and the time that the task was started. A vector of all unique hostnames in the resulting data is created, which is then used as the basis of an operation to number each individual task per hostname, resulting in a list of lists that is unpacked and appended to the data. The code for function "assign_tr_task" is given below.

```
#returns task_no when totalrender_data is ordered for each virtual machine task
assign_tr_task = function(hostn) {
  host_no = sum(totalrender_data$hostname == hostn)
  return(1:host_no)
}
```

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 take the same steps to transform this into a more meaningful and computationally compatible variable. We should also remove the duplicates from the data. With the "timestamp" values now of class POSIXct and duplicates removed, we can turn out attention to the outliers in the data. Whether these outliers should be retained in the data or not is dependent upon if they are meaningful or an error. We can now arrange gpu by the newly transformed variable "timestamp" and "hostname". The resulting data provides several new insights about 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 0 values in "gpu" are related to the periods on the virtual machine in between tasks. This can be best depicted using a plot of gpuMemUtilPerc vs Time.



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 GPU measurements whilst a task is being performed, clearly 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, as well as other performance metrics of these periods.

First, we can prepare the data that will be used in conjunction with "totalrender_data" and "task.x.y" to evaluate the task rendering performance across various metrics. In order to join these data sets for the analysis we will create the variable "task_no" that will assign a task number to each observation within a single peak in our plot above 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", that identifies a new task, or peak, 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" that groups the data by the hostname and task_no, then summarises by the mean of the different quantitative variables in the data.

```
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 any cleaning of this data.

Integrate Data

With each data set reformatted and transformed as desired, we can commence merging them to derive additional interpretations from the data. To begin, we can combine our summarised data from "gpu_summary" with "totalrender_data". Before joining these data sets we can verify that the hostname and task_no are our 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 we can join these data sets 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 pixel 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"))
```

This

Modelling

Create several graphs to view different interactions of data set

Which event types dominate task runtimes? (app checkpoints)

What is the interplay between GPU temperature and performance? (gpu)

What is the interplay between increased power draw and render time? – Can we quantify the variation in computation requirements for particular tiles?

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

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

Evaluation

Deployment