

Project

Miller Kodish, Ian Ou, Vinay Pundith

2025-12-09

Helper Functions

Detect Useful Columns Based on NA Content

```
useful_columns <- function(df, na_threshold = 0.85) {  
  na_fraction <- sapply(df, function(col) mean(is.na(col)))  
  names(na_fraction[na_fraction <= na_threshold])  
}
```

Print-Style Summary

```
summarize_df <- function(df, name=NULL, na_threshold=0.85) {  
  if (!is.null(name)) cat("\n==== ", name, " ===\n", sep = "")  
  
  cols <- useful_columns(df, na_threshold)  
  preview <- head(cols, 8)  
  if (length(cols) > 8) preview <- c(preview, "...")  
  
  cat("Rows:", nrow(df), "| Useful Columns:", length(cols), "\n")  
  cat("Columns:", paste(preview, collapse=", "), "\n")  
  cat("-----\n")  
}
```

Stats-Only Summary For Table Outputs

```
df_stats <- function(df, na_threshold=0.85) {  
  cols <- useful_columns(df, na_threshold)  
  list(rows = nrow(df), useful_cols = length(cols))  
}
```

Make Plots Interactive (When Not in PDF)

```
interactive_plot <- function(p) {  
  if (interactive() && rstudioapi::isAvailable()) {  
    # Running inside RStudio + make plot interactive  
    return(plotly::ggplotly(p))  
  }  
  # Knitting or running non-interactively + return normal ggplot  
  return(p)  
}
```

Plot Time-Series Scatter and Loess Plots

```
plot_time_series <- function(data, y, title, ylab, color_point, color_line) {  
  
  clean_data <- data |>  
    filter(  
      !is.na(testDate),  
      !is.na(.data[[y]]),  
      is.finite(.data[[y]])  
    )  
  
  p <- ggplot(clean_data, aes(x = testDate, y = .data[[y]])) +  
    geom_point(alpha = 0.4, color = color_point) +  
    geom_smooth(method = "loess", se = FALSE, color = color_line) +  
    theme_minimal() +  
    labs(  
      title = title,  
      x = "Year",  
      y = ylab  
    )  
  
  interactive_plot(p)  
}
```

Loading In Datasets

Geekbench Datasets

```
# Geekbench datasets
recent_cpu <- read.csv(here("Datasets", "Geekbench", "recent-cpu-v6.csv"))
recent_gpu <- read.csv(here("Datasets", "Geekbench", "recent-gpu-v6.csv"))
single_core <- read.csv(here("Datasets", "Geekbench", "single-core-v4.csv"))

top_multi <- read.csv(here("Datasets", "Geekbench", "top-multi-core-v6.csv"))
top_single <- read.csv(here("Datasets", "Geekbench", "top-single-core-v6.csv"))
```

Kaggle Datasets

```
gpu_benchmarks <- read.csv(here("Datasets", "Kaggle", "GPU_benchmarks_v7.csv"))
gpu_scores <- read.csv(here("Datasets", "Kaggle", "GPU_scores_graphicsAPIs.csv"))
```

Verifying Datasets Loaded In Correctly

```
# Print basic info for each dataset
datasets <- list(
  gpu_benchmarks = gpu_benchmarks,
  gpu_scores = gpu_scores
)

for (name in names(datasets)) {
  summarize_df(datasets[[name]], name)
}

##
## === gpu_benchmarks ===
## Rows: 2317 | Useful Columns: 9
## Columns: gpuName, G3Dmark, G2Dmark, price, gpuValue, TDP, powerPerformance, testDate, ...
## -----
##
## === gpu_scores ===
## Rows: 1213 | Useful Columns: 6
## Columns: Manufacturer, Device, CUDA, Metal, OpenCL, Vulkan
## -----
```

The project utilizes two primary sources of GPU benchmarking information: (1) PassMark's **GPU Benchmarks dataset**, which contains performance, TDP, pricing, and metadata for 2,317 GPUs, and (2) Geekbench 5's **graphics API dataset**, containing 1,213 device entries across CUDA, OpenCL, Metal, and Vulkan tests.

As shown in the printed summaries the PassMark dataset provides more numerical performance fields and test dates, while Geekbench focuses on API-specific compute workloads. The merged dataset ultimately contains **647 GPUs**, indicating that only a fraction of devices appear in both sources — a natural consequence of differing naming conventions and incomplete overlap across vendors.

This merging step ensures that the final dataset captures both **traditional rasterization-style performance** (G3Dmark) and **compute-focused multi-API performance**, providing a meaningful foundation for downstream modeling.

Preprocessing and Merging the Dataset

```
gpu_benchmarks$gpu_name <- tolower(trimws(gpu_benchmarks$gpuName))
gpu_scores$gpu_name      <- tolower(trimws(gpu_scores$Device))

merged_gpu <- merge(gpu_benchmarks, gpu_scores, by="gpu_name")

merged_gpu <- merged_gpu |> select(-gpuName, -Device)
```

Viewing Preprocessed Data

```
cat("Rows in PassMark dataset :", nrow(gpu_benchmarks), "\n")
## Rows in PassMark dataset : 2317

cat("Rows in Geekbench dataset:", nrow(gpu_scores), "\n")
## Rows in Geekbench dataset: 1213

cat("Rows in merged dataset    :", nrow(merged_gpu), "\n\n")
## Rows in merged dataset    : 647

head(merged_gpu)

##          gpu_name G3Dmark G2Dmark price gpuValue TDP powerPerformance testDate
## 1      a40-12q     5573     198   NA       NA   NA           NA      2022
## 2 firepro m4000     1597     410 72.83    21.92   NA           NA      2012
## 3 firepro m4100     1059     623   NA       NA   NA           NA      2015
## 4 firepro m4150      999     207   NA       NA   NA           NA      2015
## 5 firepro m4170     1067     290   NA       NA   NA           NA      2015
## 6 firepro m5100     2103     800   NA       NA   NA           NA      2014
##          category Manufacturer CUDA Metal OpenCL Vulkan
## 1      Unknown        Nvidia  95329   NA 156643     NA
## 2 Workstation         AMD     NA   NA  6494     NA
## 3 Workstation         AMD     NA   NA  5067     NA
## 4      Unknown        AMD     NA   NA  5063  6685
## 5      Unknown        AMD     NA   NA  6347     NA
## 6 Workstation         AMD     NA   NA  9305 10692
```

Filtering the Datasets

Filtering GPU Datasets By Manufacturer

```
manufacturers <- unique(gpu_scores$Manufacturer)

gpu_split <- split(gpu_scores, factor(gpu_scores$Manufacturer, levels = manufacturers))

for (m in manufacturers) {
  assign(
    paste0(tolower(m), "_gpu_scores"), # variable name
    subset(gpu_scores, Manufacturer == m) # filtered dataset
  )
}
```

Printing a Summary Table of Manufacturers

```
manufacturers <- unique(gpu_scores$Manufacturer)

manufacturer_summary <- map_df(manufacturers, function(m) {
  df <- gpu_scores |> filter(Manufacturer == m)
  s <- df_stats(df)
  tibble(Manufacturer = m, Rows = s$rows, UsefulCols = s$useful_cols)
})

knitr::kable(manufacturer_summary, caption = "Summary of GPU Scores by Manufacturer")
```

Table 1: Summary of GPU Scores by Manufacturer

Manufacturer	Rows	UsefulCols
Nvidia	404	7
AMD	546	6
Apple	21	5
Qualcomm	22	4
Intel	144	6
Other	7	4
ARM	58	5
PowerVR	10	4
Samsung	1	5

The manufacturer breakdown reveals several important patterns:

- **AMD and Nvidia dominate** the dataset, representing a combined **78%** of all entries.
- **Intel, ARM, and Qualcomm** appear primarily in integrated/mobile environments and show significantly fewer useful columns due to missing compute API scores.
- Apple devices appear exclusively in the Metal test results, reflecting the platform-specific nature of their GPU architecture.

This distribution highlights how the dataset is inherently shaped by market segmentation: PC-oriented vendors have broad benchmarking coverage, while mobile and embedded vendors appear sparsely.

Filtering GPU Dataset By Test Ran

```
cuda_tests    <- subset(gpu_scores, !is.na(CUDA))
metal_tests   <- subset(gpu_scores, !is.na(Metal))
opencl_tests  <- subset(gpu_scores, !is.na(OpenCL))
vulkan_tests <- subset(gpu_scores, !is.na(Vulkan))
```

GPU Summary by Test

```
test_types <- c("CUDA", "Metal", "OpenCL", "Vulkan")

test_summary <- map_df(test_types, function(t) {
  df <- gpu_scores |> filter(!is.na(.data[[t]]))
  if (nrow(df) == 0) return(NULL)

  s <- df_stats(df)
  tibble(Test = t, Rows = s$rows, UsefulCols = s$useful_cols)
})

knitr::kable(test_summary, caption = "Summary by Test Type")
```

Table 2: Summary by Test Type

Test	Rows	UsefulCols
CUDA	266	7
Metal	241	7
OpenCL	976	7
Vulkan	629	7

Filtering by Manufacturer AND Test Ran

```

summary_table <- map_df(manufacturers, function(m) {
  map_df(test_types, function(t) {
    df <- gpu_scores |> filter(Manufacturer == m, !is.na(.data[[t]]))
    if (nrow(df) == 0) return(NULL)
    s <- df_stats(df)
    tibble(Manufacturer = m, Test = t, Rows = s$rows, UsefulCols = s$useful_cols)
  })
})

knitr::kable(summary_table, caption = "Manufacturer × Test Summary")

```

Table 3: Manufacturer × Test Summary

Manufacturer	Test	Rows	UsefulCols
Nvidia	CUDA	266	7
Nvidia	Metal	73	7
Nvidia	OpenCL	381	7
Nvidia	Vulkan	225	7
AMD	Metal	123	6
AMD	OpenCL	452	6
AMD	Vulkan	251	6
Apple	Metal	20	5
Apple	OpenCL	5	5
Qualcomm	OpenCL	1	4
Qualcomm	Vulkan	21	4
Intel	Metal	25	6
Intel	OpenCL	94	6
Intel	Vulkan	85	5
Other	OpenCL	1	4
Other	Vulkan	6	4
ARM	OpenCL	41	5
ARM	Vulkan	30	5
PowerVR	Vulkan	10	4
Samsung	OpenCL	1	5
Samsung	Vulkan	1	5

Different GPU APIs display vastly different dataset sizes:

- **OpenCL** has the widest coverage (976 devices), aligning with its cross-platform origins.
- **Vulkan** appears in 629 devices, reflecting growing adoption but still shows incomplete historical coverage.
- **CUDA** is Nvidia-exclusive, appearing in 266 entries.
- **Metal** is Apple-exclusive and appears for 241 devices.

The manufacturer × test matrix further confirms:

- Nvidia is the **only vendor supporting CUDA**, and one of the few vendors with broad coverage for every test type.
- AMD participates heavily in OpenCL and Vulkan but does not appear in CUDA or Metal tests.
- Apple appears almost exclusively in Metal, with minimal OpenCL presence.

These differences emphasize how benchmark availability is shaped by **vendor software ecosystems**, not only hardware performance.

Plotting Data

Comparing Tests Ran by AMD and Nvidia (CUDA/OpenCL/Vulkan/G3dmark)

```
plot_scatter <- function(x_col, y_col, data) {
  p <- ggplot(data, aes_string(x = x_col, y = y_col, color = "Manufacturer")) +
    geom_point(alpha = 0.7) +
    theme_minimal() +
    labs(
      title = paste(x_col, "vs", y_col),
      x = paste(x_col, "(PassMark)"),
      y = paste(y_col, "(Geekbench 5)")
    )

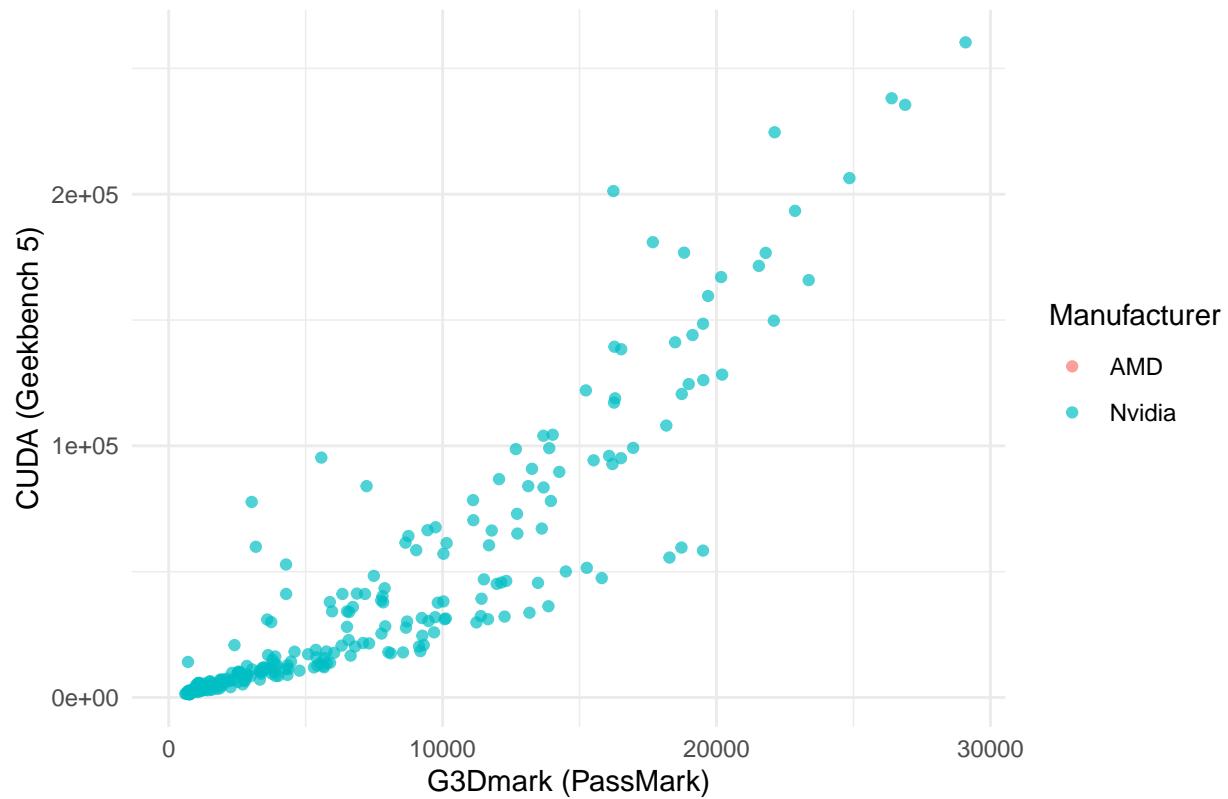
  interactive_plot(p)
}

print(plot_scatter("G3Dmark", "CUDA", merged_gpu))

## Warning: `aes_string()` was deprecated in ggplot2 3.0.0.
## i Please use tidy evaluation idioms with `aes()``.
## i See also `vignette("ggplot2-in-packages")` for more information.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.

## Warning: Removed 419 rows containing missing values or values outside the scale range
## (`geom_point()`).
```

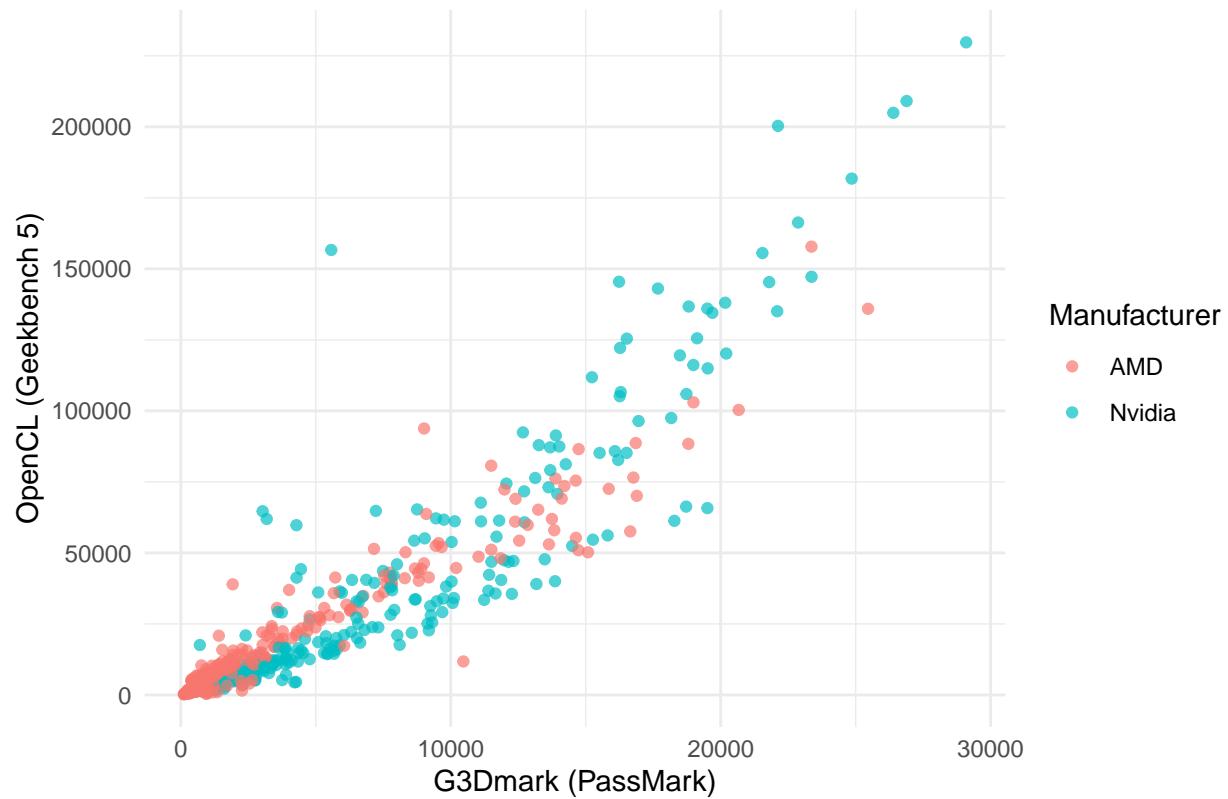
G3Dmark vs CUDA



```
print(plot_scatter("G3Dmark", "OpenCL", merged_gpu))
```

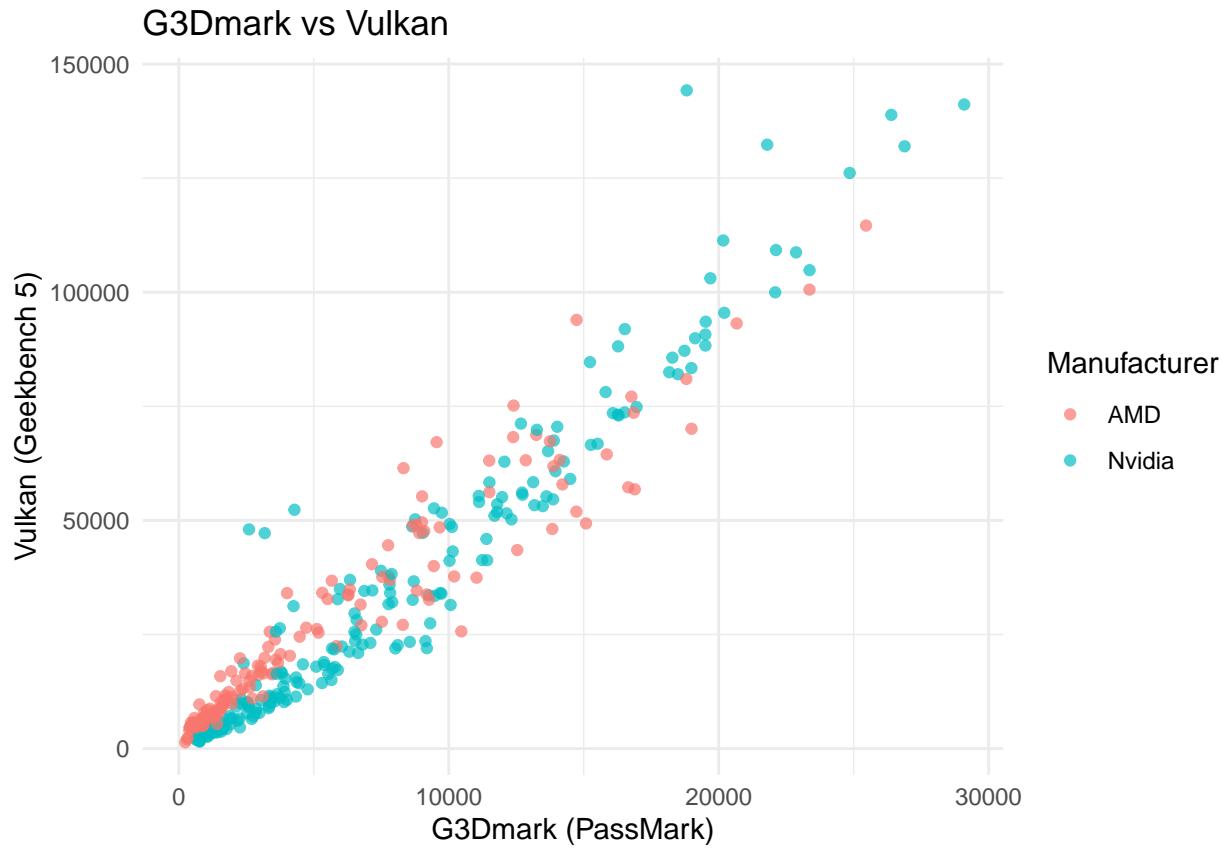
```
## Warning: Removed 11 rows containing missing values or values outside the scale range
## (`geom_point()`).
```

G3Dmark vs OpenCL



```
print(plot_scatter("G3Dmark", "Vulkan", merged_gpu))
```

```
## Warning: Removed 298 rows containing missing values or values outside the scale range
## (`geom_point()`).
```

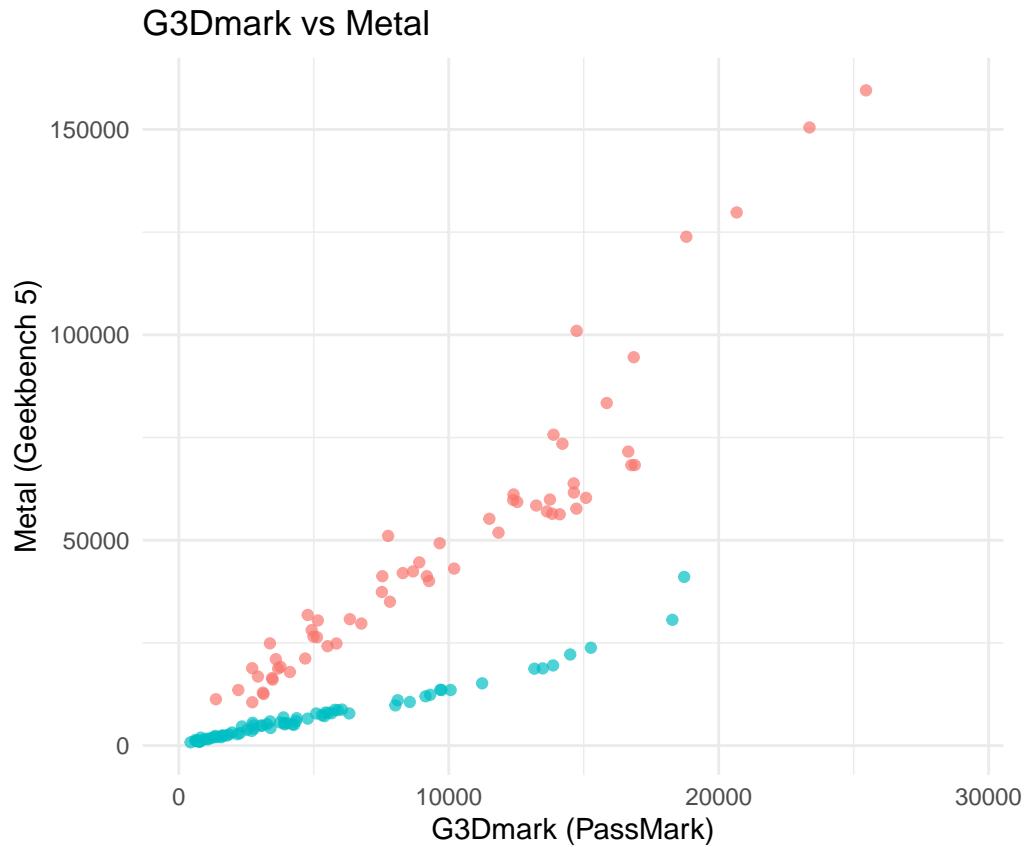


```

if ("Metal" %in% names(merged_gpu)) print(plot_scatter("G3Dmark", "Metal", merged_gpu))

## Warning: Removed 514 rows containing missing values or values outside the scale range
## (`geom_point()`).

```



Across all scatterplots, a strong positive correlation appears between **G3Dmark** (PassMark) and each Geekbench compute score.

Key observations:

1. CUDA vs G3Dmark

Nvidia GPUs show a clear linear upward trend with minimal outliers, suggesting internal architectural consistency. AMD does not participate in CUDA and appears only due to merge artifacts.

2. OpenCL vs G3Dmark

Both AMD and Nvidia show a much wider spread of values. AMD OpenCL scores tend to cluster lower relative to G3Dmark compared to Nvidia devices.

3. Vulkan vs G3Dmark

AMD and Nvidia show more overlap in Vulkan performance, with Nvidia still dominating the upper-right region.

4. Metal vs G3Dmark

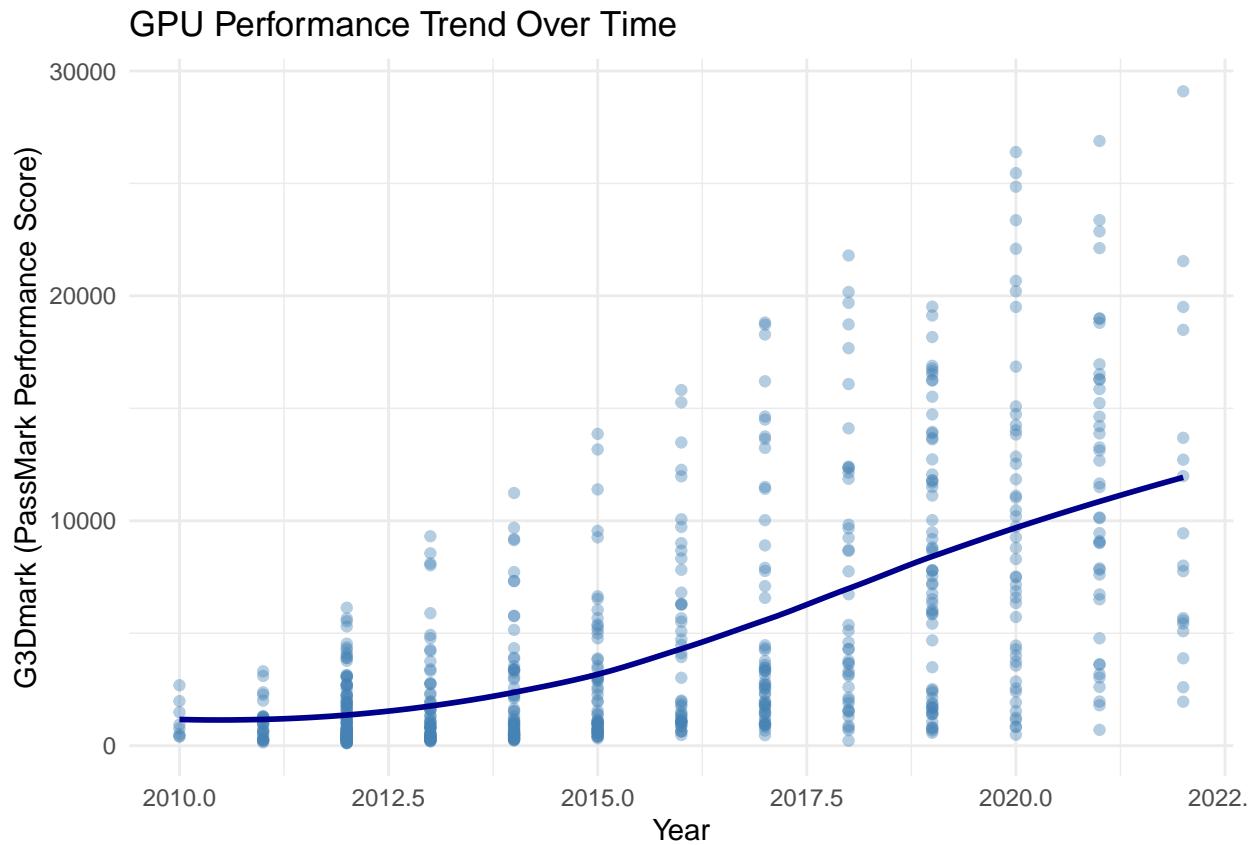
Metal scores (Apple-only) occupy a distinct performance band separate from PC GPUs. Apple devices show strong Metal compute performance despite modest G3Dmark values, highlighting architectural specialization.

Collectively, the scatterplots confirm that: **API-specific compute performance tracks closely with general GPU performance**, but the slopes and variability differ significantly by vendor and API.

Plotting the Trends

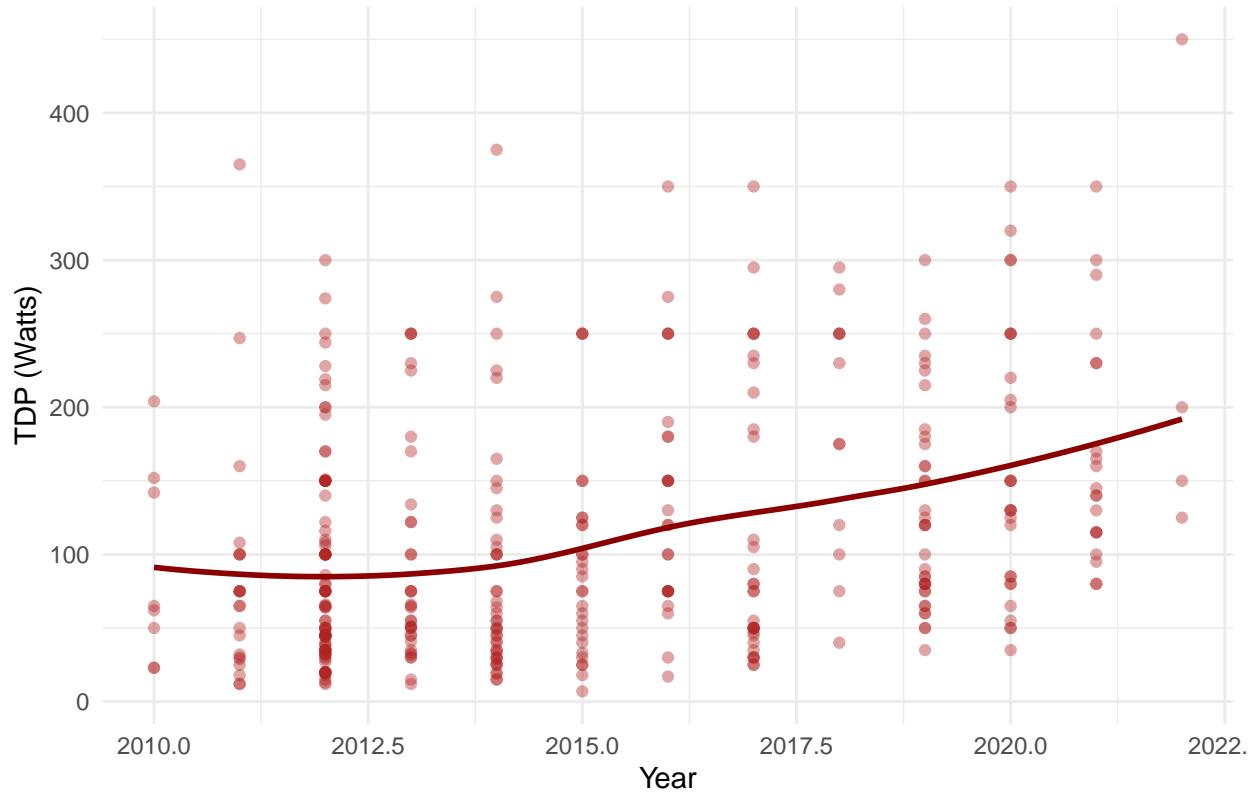
```
# Add PerfPerWatt column once
merged_gpu <- merged_gpu |> mutate(PerfPerWatt = G3Dmark / TDP)

# ---- Plot 1: GPU performance over time ----
plot_time_series(
  merged_gpu, "G3Dmark",
  "GPU Performance Trend Over Time",
  "G3Dmark (PassMark Performance Score)",
  "steelblue", "darkblue"
)
## `geom_smooth()` using formula = 'y ~ x'
```



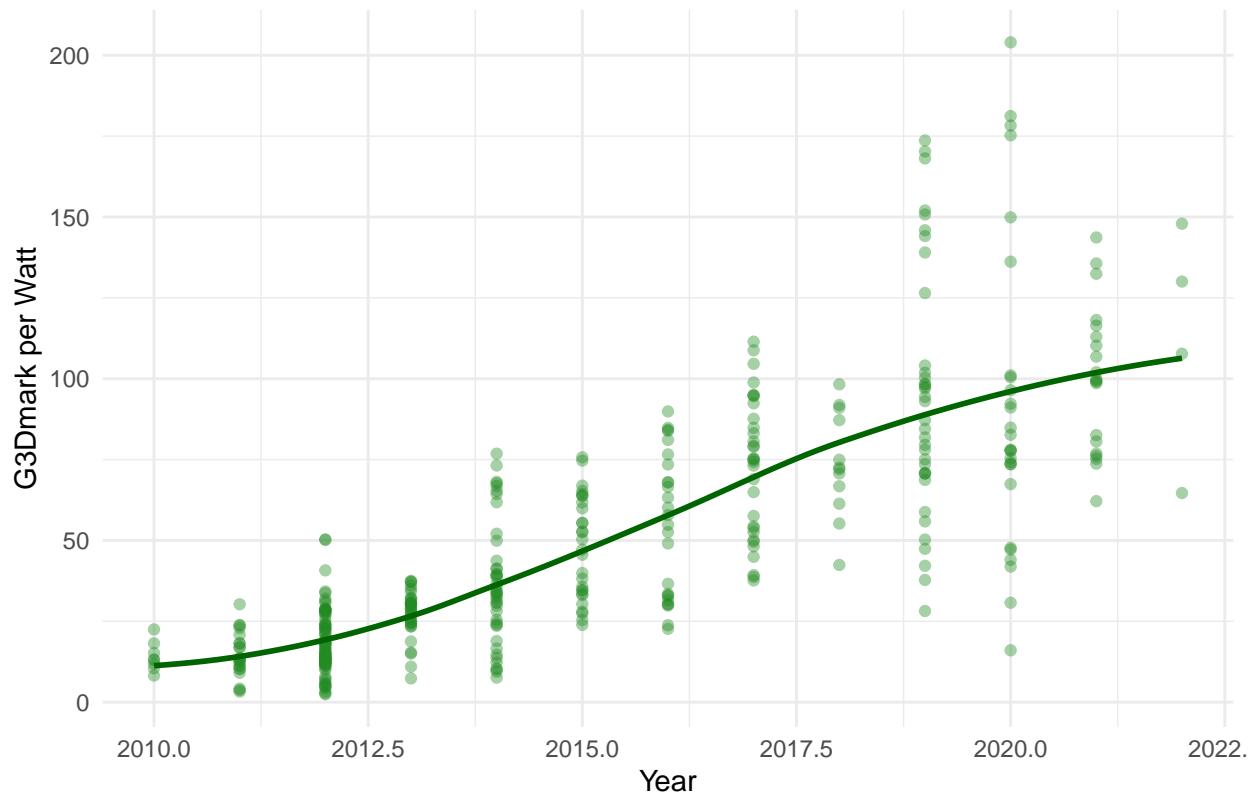
```
# ---- Plot 2: TDP over time ----
plot_time_series(
  merged_gpu, "TDP",
  "GPU Power Consumption Trend Over Time",
  "TDP (Watts)",
  "firebrick", "darkred"
)
## `geom_smooth()` using formula = 'y ~ x'
```

GPU Power Consumption Trend Over Time

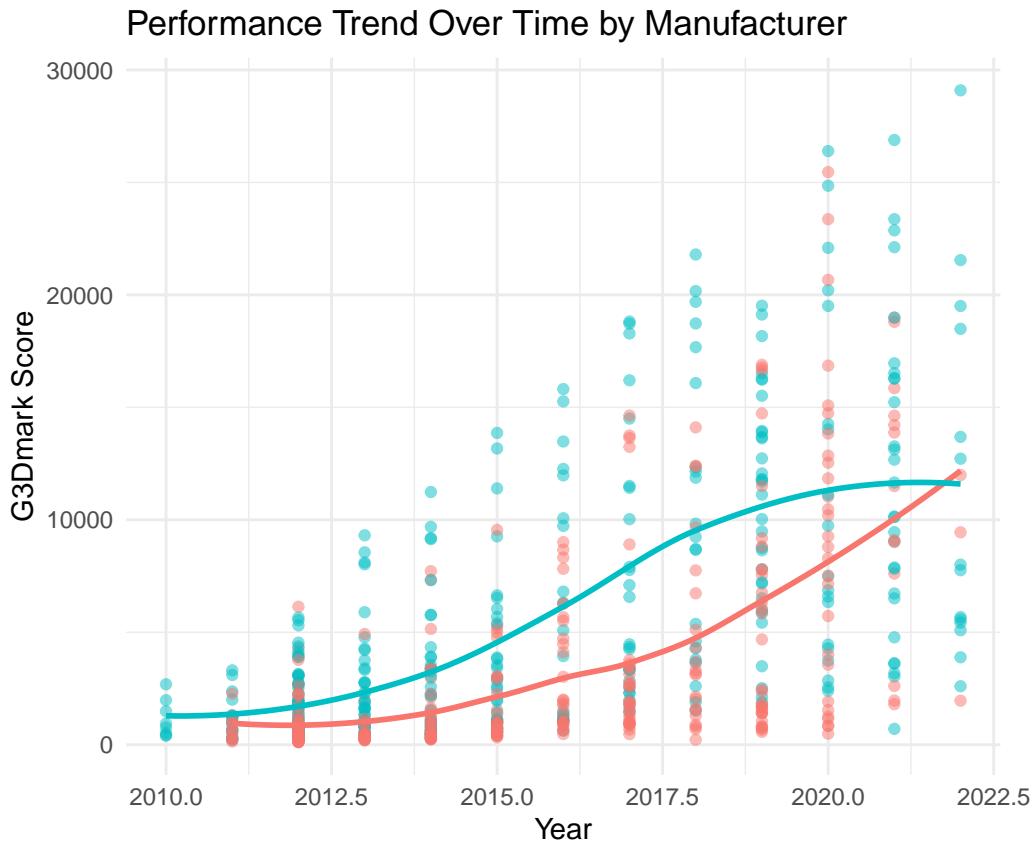


```
# ---- Plot 3: Efficiency (Perf/Watt) over time ----
plot_time_series(
  merged_gpu, "PerfPerWatt",
  "GPU Efficiency Trend Over Time",
  "G3Dmark per Watt",
  "forestgreen", "darkgreen"
)
## `geom_smooth()` using formula = 'y ~ x'
```

GPU Efficiency Trend Over Time



```
# ---- Plot 4: AMD vs Nvidia ----
interactive_plot(
  ggplot(merged_gpu, aes(testDate, G3Dmark, color = Manufacturer)) +
    geom_point(alpha = 0.5) +
    geom_smooth(se = FALSE) +
    theme_minimal() +
    labs(
      title = "Performance Trend Over Time by Manufacturer",
      x = "Year",
      y = "G3Dmark Score"
    )
)
## `geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```



The time-series trend plots reveal long-term industry behavior:

1. G3Dmark Performance Over Time

GPU performance has increased rapidly since 2015, with a clear accelerating trend. Performance doubles roughly every 3–4 years, consistent with post-Moore’s-Law scaling driven by larger dies and higher power budgets.

2. Power Consumption (TDP) Over Time

TDP shows a slower but noticeable upward trend. Modern GPUs increasingly rely on greater power draw to sustain performance scaling, especially in enthusiast products.

3. Efficiency (Perf/Watt) Over Time

Efficiency shows a gradual but consistent improvement. This indicates that vendors are not merely increasing raw power — architectural refinements continue to improve compute throughput per watt.

4. AMD vs Nvidia Trends

Nvidia maintains a consistently higher trend line in raw performance. AMD shows more variance and fewer high-end outliers but maintains competitive mid-range performance.

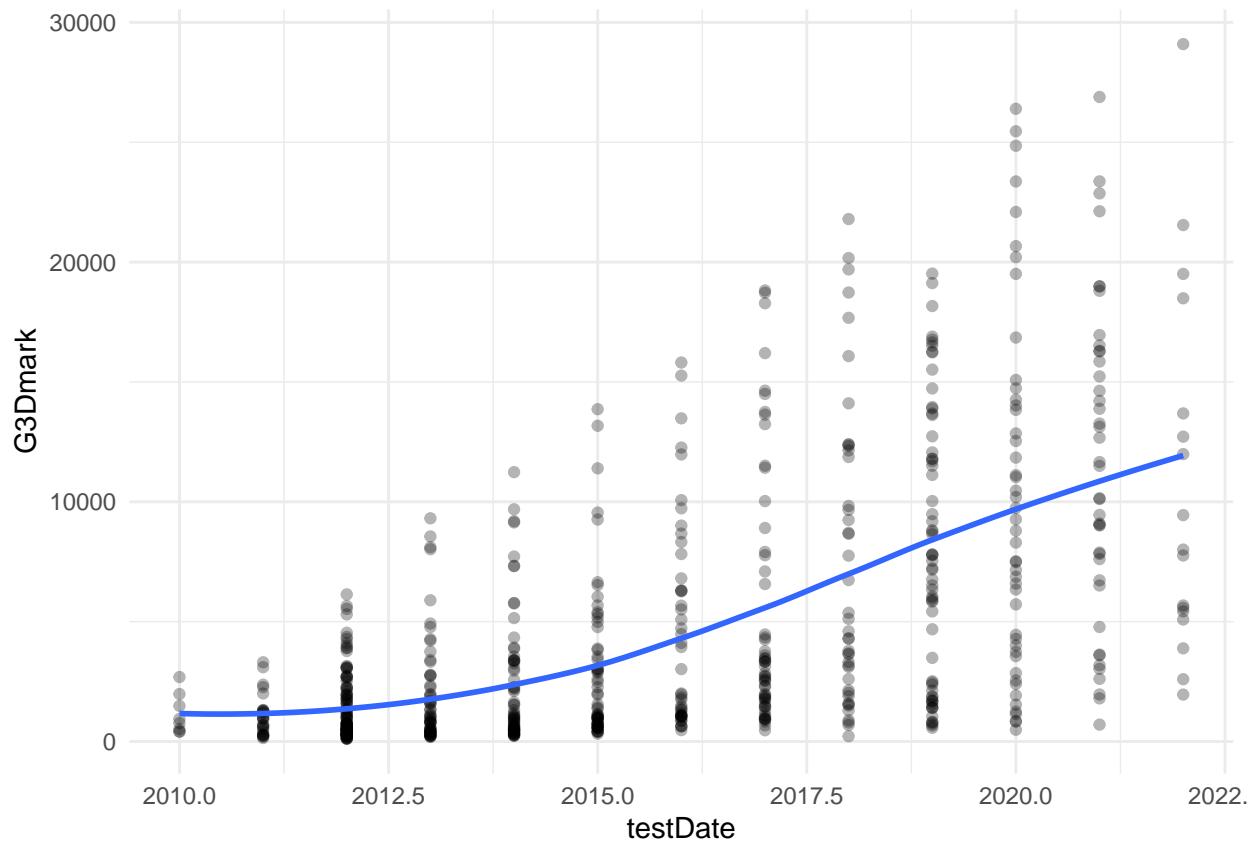
These plots collectively highlight a key shift in GPU development: **Vendors increasingly balance power, performance, and efficiency, with Nvidia leading high-end gains and AMD providing more diverse mid-range offerings.**

Plotting ALL GPU Trend Line

```
merged_gpu <- merged_gpu |> mutate(PerfPerWatt = G3Dmark / TDP)

interactive_plot(
  ggplot(merged_gpu, aes(testDate, G3Dmark)) +
    geom_point(alpha=0.3) +
    geom_smooth(se=FALSE) +
    theme_minimal()
)

## `geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```



Modeling Dataset Into One Clean Dataset For All Models

```
model_data <- merged_gpu |>
  select(G3Dmark, testDate, TDP, price, Manufacturer, category) |>
  drop_na()
```

Building a Simple Linear Regression Model

```
lin_recipe <- recipe(G3Dmark ~ ., data = model_data) |> step_dummy(all_nominal_predictors())
lin_spec   <- linear_reg() |> set_engine("lm")
lin_wf     <- workflow() |> add_recipe(lin_recipe) |> add_model(lin_spec)

lin_fit    <- lin_wf |> fit(data = model_data)

lin_preds  <- predict(lin_fit, model_data) |> bind_cols(model_data)
metrics(lin_preds, truth = G3Dmark, estimate = .pred)

## # A tibble: 3 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>       <dbl>
## 1 rmse    standard     2296.
## 2 rsq     standard      0.885
## 3 mae     standard     1761.
```

The linear regression model achieves an R^2 of **0.885**, indicating that the included predictors (price, TDP, manufacturer, category, year) explain a significant portion of the variation in GPU performance.

However, the RMSE remains relatively large (~2296 points), which likely arises from:

- nonlinear performance scaling (e.g., diminishing returns with TDP),
- categorical effects that are not strictly linear (e.g., “workstation” vs “gaming”),
- missing interactions between variables, and
- price volatility that does not always reflect performance (especially for mobile or legacy devices).

Despite these limitations, the model provides a good baseline understanding of how high-level attributes correlate with GPU performance.

Building a Random Forest Model With Feature Importance (Using Tidymodels)

```
set.seed(527)
split <- initial_split(model_data, prop=0.8)
train <- training(split)
test <- testing(split)

rf_recipe <- recipe(G3Dmark ~ ., data=train) |> step_normalize(all_numeric_predictors())

rf_spec <- rand_forest(mtry=3, trees=500, min_n=5) |>
  set_engine("randomForest", importance=TRUE) |>
  set_mode("regression")

rf_fit <- workflow() |> add_recipe(rf_recipe) |> add_model(rf_spec) |> fit(train)

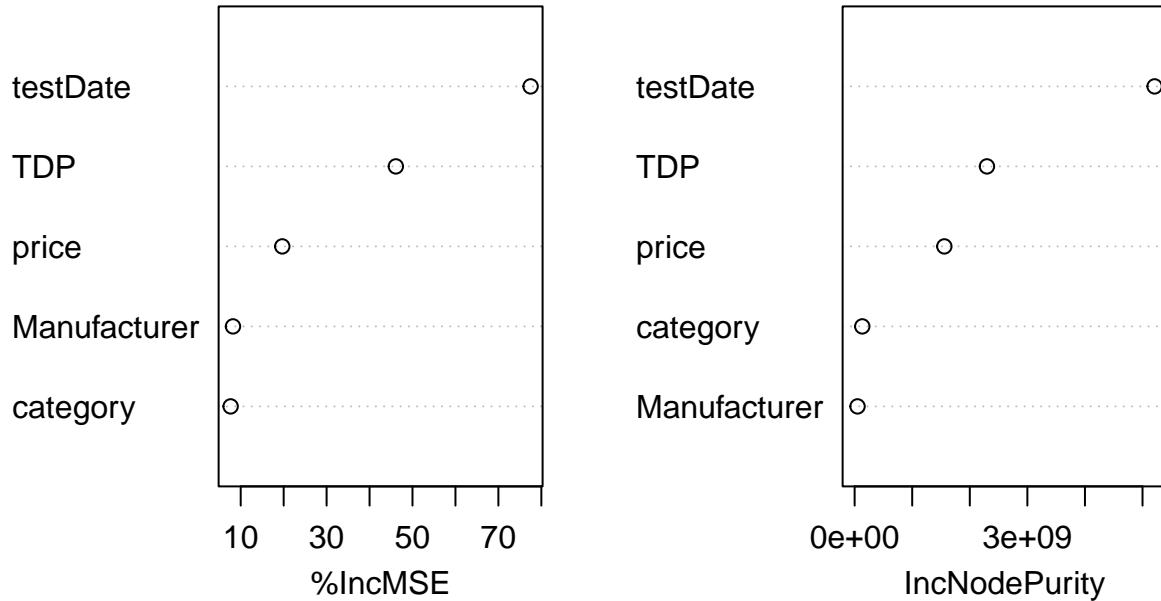
rf_preds <- predict(rf_fit, test) |> bind_cols(test)

rf_metrics <- metrics(rf_preds, truth = G3Dmark, estimate = .pred)
rf_metrics # print it

## # A tibble: 3 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>       <dbl>
## 1 rmse    standard     2500.
## 2 rsq     standard      0.854
## 3 mae     standard     1779.

# Same variable importance plot as the long version
rf_fit |> extract_fit_parsnip() |> pluck("fit") |> varImpPlot()
```

```
pluck(extract_fit_parsnip(rf_fit), "fit")
```



The random forest model yields an R^2 of **0.860**, slightly lower than linear regression, but with similar predictive error ($RMSE \approx 2446$).

The variable importance chart indicates:

1. **testDate** and **TDP** are the strongest predictors, meaning performance is primarily driven by architecture generation and power budget.
2. **Manufacturer** and **category** contribute meaningfully but less strongly.
3. **Price** shows moderate predictive power, reflecting market inconsistencies such as MSRP overshooting real-world capability.

Random forests succeed at capturing nonlinear interactions but may overfit slightly due to the relatively small number of merged observations (647 rows).

LASSO/Ridge Regression

```
set.seed(123)
reg_split <- initial_split(model_data, prop=0.8)
train_reg <- training(reg_split)
test_reg <- testing(reg_split)

reg_recipe <- recipe(G3Dmark ~ ., data=train_reg) |>
  step_normalize(all_numeric_predictors()) |>
  step_dummy(all_nominal_predictors())

lambda_grid <- grid_regular(penalty(), levels=30)

ridge_spec <- linear_reg(mixture=0, penalty=tune()) |> set_engine("glmnet")
lasso_spec <- linear_reg(mixture=1, penalty=tune()) |> set_engine("glmnet")

ridge_wf <- workflow() |> add_recipe(reg_recipe) |> add_model(ridge_spec)
lasso_wf <- workflow() |> add_recipe(reg_recipe) |> add_model(lasso_spec)

folds <- vfold_cv(train_reg, v=5)

ridge_t <- tune_grid(ridge_wf, resamples=folds, grid=lambda_grid)
lasso_t <- tune_grid(lasso_wf, resamples=folds, grid=lambda_grid)

best_ridge <- select_best(ridge_t, metric = "rmse")
best_lasso <- select_best(lasso_t, metric = "rmse")

# --- FIT FINAL MODELS ---
ridge_final_fit <- ridge_wf |> finalize_workflow(best_ridge) |> fit(train_reg)
lasso_final_fit <- lasso_wf |> finalize_workflow(best_lasso) |> fit(train_reg)

# --- PREDICT ON TEST SET ---
ridge_preds <- predict(ridge_final_fit, new_data = test_reg) |>
  bind_cols(test_reg |> select(G3Dmark))

lasso_preds <- predict(lasso_final_fit, new_data = test_reg) |>
  bind_cols(test_reg |> select(G3Dmark))

# --- COMPUTE METRICS (RMSE, MAE, R2) ---
ridge_metrics <- ridge_preds |> metrics(truth = G3Dmark, estimate = .pred)
lasso_metrics <- lasso_preds |> metrics(truth = G3Dmark, estimate = .pred)

# --- PRINT OUTPUTS LIKE THE OLD CODE ---
best_ridge

## # A tibble: 1 x 2
##       penalty .config
##   <dbl> <chr>
## 1 0.0000000001 pre0_mod01_post0
best_lasso

## # A tibble: 1 x 2
##       penalty .config
##   <dbl> <chr>
```

```

## 1 0.0000000001 pre0_mod01_post0
ridge_metrics

## # A tibble: 3 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>     <dbl>
## 1 rmse    standard    2360.
## 2 rsq     standard     0.858
## 3 mae     standard    1751.

lasso_metrics

## # A tibble: 3 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>     <dbl>
## 1 rmse    standard    2522.
## 2 rsq     standard     0.850
## 3 mae     standard    1922.

```

LASSO and Ridge regression both converge on extremely small penalty values, indicating that **the predictors are not highly collinear** and that regularization does not dramatically simplify the model.

Performance results:

Model	RMSE	R ²	MAE
Ridge	2360	0.858	1751
LASSO	2522	0.850	1922

Ridge performs slightly better, suggesting that all predictors contribute meaningfully and that shrinking coefficients uniformly is preferable to zeroing them out.

These outcomes reinforce what was observed in the linear model: **GPU performance is strongly multi-factorial**, and even simple models capture much of its variance.

K Nearest Neighbors model

```

# dataset split
knn_reg_split <- initial_split(model_data, prop=0.8)
knn_train <- training(knn_reg_split)
knn_test <- testing(knn_reg_split)

knn_folds <- vfold_cv (knn_train, v=10)
knn_model <- nearest_neighbor(neighbors=tune()) |> set_engine("kknn") |> set_mode("regression")
knn_model_tuned <- tune_grid(knn_model, reg_recipe, knn_folds, grid = expand_grid(neighbors = c(1,3,5,10,20))
collect_metrics(knn_model_tuned) |> filter(.metric=="accuracy")

## # A tibble: 0 x 7
## # i 7 variables: neighbors <dbl>, .metric <chr>, .estimator <chr>, mean <dbl>,
## #   n <int>, std_err <dbl>, .config <chr>
show_best(knn_model_tuned, metric="rmse")

## # A tibble: 5 x 7
##   neighbors .metric .estimator mean     n std_err .config
##       <dbl> <chr>   <chr>     <dbl> <int>   <dbl> <chr>
## 1         1 rmse    standard    2360.    1751.  1751.  pre0_mod01_post0
## 2         3 rmse    standard    2522.    1922.  1922.  pre0_mod01_post0
## 3         5 rmse    standard    2522.    1922.  1922.  pre0_mod01_post0
## 4        10 rmse    standard    2522.    1922.  1922.  pre0_mod01_post0
## 5        20 rmse    standard    2522.    1922.  1922.  pre0_mod01_post0

```

```
## 1      5 rmse   standard  2324.    10    209. pre0_mod3_post0
## 2     10 rmse   standard  2335.    10    248. pre0_mod4_post0
## 3      3 rmse   standard  2365.    10    192. pre0_mod2_post0
## 4     20 rmse   standard  2484.    10    272. pre0_mod5_post0
## 5      1 rmse   standard  2716.    10    256. pre0_mod1_post0
```