Performance Analysis on Modern CPUs

Exercise 1

In this exercise, we will familiarize ourselves with a few system monitoring tools and the **`perf`** profiler by analyzing performance of a ROOT tutorial. The tutorial we will use implements a simplified yet complex example of a physics analysis, which reconstructs the Higgs boson decaying to two Z bosons from events with four leptons. The data and simulated events are taken from CERN open data representing a subset of the data recorded in 2012 with the CMS detector at the LHC. The tutorial follows the Higgs to four leptons analysis published on CERN Open Data portal (10.7483/OPENDATA.CMS.JKB8.RR42). The resulting plots show the invariant mass of the selected four lepton systems in different decay modes (four muons, four electrons and two of each kind) and in a combined plot indicating the decay of the Higgs boson with a mass of about 125 GeV. The tutorial can be found in ROOT's website at the URL below:

https://root.cern/doc/master/df103__NanoAODHiggsAnalysis_8C.html

1 Setup

In this exercise, we will use software from CVMFS. Let's start by setting up the stack `LCG_108` for Alma Linux 9 built with GCC 14 with frame pointers enabled. For your convenience, you may copy the source command from the listing below:

```
$ source /cvmfs/sft.cern.ch/lcg/views/LCG_108/x86_64-el9-gcc14fp-opt/setup.sh
$ command -v root && root --version
/cvmfs/sft.cern.ch/lcg/views/LCG_108/x86_64-el9-gcc14fp-opt/bin/root
ROOT Version: 6.36.02
Built for linuxx8664gcc on Jul 31 2025, 15:05:53
From tags/6-36-02@6-36-02
```

```
bash ~ $ ssh csc-2025-02.cern.ch
* ------------------------------------------------------------------------
* CERN School of Computing - 2025
* hostgroup: - csc/cc/pm, env: - qa, admins: csc-cc-admins@cern.ch
* Availability zone: none
* Please set a host or hostgroup parameter 'comment' to describe your host or hostgroup.
* ------------------------------------------------------------------------
Last login: Mon Sep 29 14:27:21 2025 from 128.141.150.247
csc-2025-02 ~ $ source /cvmfs/sft.cern.ch/lcg/views/LCG_108/x86_64-el9-gcc14fp-opt/setup.sh
csc-2025-02 ~ $ command -v gcc
/cvmfs/sft.cern.ch/lcg/releases/gcc/14.2.0.fp-6ad5e/x86_64-el9/bin/gcc
csc-2025-02 ~ $ gcc -v 2>&1 | grep enable-frame-pointer
Configured with: /workspace/build_fp/contrib/gcc-14.2.0.fp/src/gcc/14.2.0.fp/configure --prefix=/workspace/lcgcmake-install/gc
c/14.2.0.fp/x86_64-el9 --disable-nls --with-system-zlib --enable-languages=c,c++,fortran,lto,go --with-gnu-ld --with-gnu-as --
enable-__cxa_atexit --disable-libunwind-exceptions --enable-gnu-unique-object --enable-plugin --with-linker-hash-style=gnu --e
nable-linker-build-id --enable-libstdcxx-time=rt --enable-shared --disable-libgcj --disable-multilib --enable-frame-pointer
csc-2025-02 ~ $ root --version
ROOT Version: 6.36.02
Built for linuxx8664gcc on Jul 31 2025, 15:05:53
From tags/6-36-02@6-36-02
csc-2025-02 ~ $ root-config --tutdir
/cvmfs/sft.cern.ch/lcg/releases/ROOT/6.36.02-c35af/x86_64-el9-gcc14fp-opt/tutorials
csc-2025-02 ~ $ ls $(root-config --tutdir)
analysis          demoshelp.C   experimental  hsimple.C     index.md          math          visualisation
CMakeLists.txt    demoshelp.py  gallery.root  hsimple.py    io                roofit
CTestCustom.cmake demos.py      geom          hsimple.root  legacy            rootlogoff.C
demos.C           evegen        hist          http          machine_learning  rootlogon.C
```

Figure 1: Setting up LCG 108 software stack from CVMFS.

The machines are setup with your home directory in AFS. However we don't want to use that. Instead, we will create a new directory in `/home` to use for the exercises.

```
$ ls /home
ops
$ mkdir -p /home/$USER/exercise1 && cd /home/$USER/exercise1
```

Now, let's copy over the ROOT tutorial we want to analyze into the current directory.

```
$ cp $(root-config --tutdir)/analysis/dataframe/df103_NanoAODHiggsAnalysis.C .
$ cp $(root-config --tutdir)/analysis/dataframe/df103_NanoAODHiggsAnalysis.py .
```

We are copying both C++ and Python versions, as we will run and analyze both of them. The tutorial can run in two modes: a normal mode which runs over the full dataset, and a fast mode, which is activated by default and runs over a skimmed dataset containing pre-filtered events. Let's copy the tutorial, modify it, and compile it, such that we can run the normal and fast modes at will later.

```
$ cp df103_NanoAODHiggsAnalysis.C df103_NanoAODHiggsAnalysis_fast.C
$ grep fast= df103_NanoAODHiggsAnalysis.C
   df103_NanoAODHiggsAnalysis(/*fast=*/true);
$ sed -i -e '/fast=/s/true/false/' df103_NanoAODHiggsAnalysis.C
$ grep fast= df103_NanoAODHiggsAnalysis.C
   df103_NanoAODHiggsAnalysis(/*fast=*/false);
$ $(root-config --cxx --cflags --libs) -O2 -g -o df103-full df103_NanoAODHiggsAnalysis.C
$ $(root-config --cxx --cflags --libs) -O2 -g -o df103-fast df103_NanoAODHiggsAnalysis_fast.C
$ ls
df103-fast  df103_NanoAODHiggsAnalysis.C       df103_NanoAODHiggsAnalysis_local.C
df103-full  df103_NanoAODHiggsAnalysis_fast.C  df103_NanoAODHiggsAnalysis.py
$ file df103-full
df103-full: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), dynamically linked, interpreter
↪  /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=846a2006d00b65b9bc7de81c3447f2f4605b2675, for GNU/Linux
↪  3.2.0, with debug_info, not stripped
```

Now we have the binaries **`df103-full`** and **`df103-fast`** which we can run to analyze performance.

2 System-level Performance Characteristics

Please check how many CPUs your machine has and take note. You can do this with **`nproc`** and **`lscpu`**, for example. How many physical and logical cores does the machine have?

```
$ nproc
$ lscpu | head -n 15
```

Run and time each version of the tutorial, as shown below.

```
$ time ./df103-fast
$ time ./df103-full
```

Now run each analysis again while watching what is happening with each of **`htop`**, **`mpstat`**, and **`perf stat`**. Is this analysis fully utilizing the CPU? Which mode has higher CPU utilization? Is the CPU utilization constant over the duration of this analysis? What is the maximum utilization that each mode achieves?

```
$ ./df103-fast & mpstat 1 8
$ ./df103-full & mpstat 1 20
$ ./df103-full >/dev/null 2>&1 & htop
$ perf stat --repeat 3 -- ./df103-fast
$ perf stat -M CPU_Utilization -I 1000 -- ./df103-full
$ perf stat -M CPU_Utilization --all-cpus -I 1000 -- ./df103-full
```

3 Networking Characteristics

Since this analysis accesses open data from EOS, let's observe how much and how fast data is being accessed. Run each analysis and check what is happening with each of **`ifstat`**, and **`bmon`**. The **`ifstat`** tool displays network statistics since last call by default, so you can run it before the analysis to reset the statistics, then afterwards to see roughly how much data was accessed via the network. The speed of the network link can be found under `/sys/class/net` for each device. It's important to know how fast your network card is to know if you are limited by network performance.

```
$ ifstat --nooutput --reset
$ ./df103-fast
$ ifstat eth0
$ cat /sys/class/net/eth0/speed
```

With **`bmon`**, you can watch what is happening in real time (first command below), adjust rates at which the information is checked and displayed, and also print output with a custom format. Finally, you may also use **`sar`** to print statistics every second.

```
$ ./df103-full >/dev/null 2>&1 & bmon --use-bit --use-si -p eth0
$ ./df103-full >/dev/null 2>&1 & bmon -p eth0 --rate-interval=0.25 --read-interval=0.25
$ ./df103-full >/dev/null 2>&1 & bmon -p eth0 -o format:fmt='$(element:name) $(attr:rxrate:bytes)\n'
$ ./df103-full >/dev/null & sar --human --iface=eth0 -n DEV 1 20
```

How much data did each of the analyses read from the network? Knowing that the full dataset is 13 GB in size, and the skimmed dataset 51 MB, what percentage of the dataset has been accessed by each analysis? What was the peak network speed reached by the full analysis? Was the performance of the full analysis limited by the network?

4 Profiling with valgrind

Let's now run a profile of this analysis at the application level. First, let's run the fast analysis with **`valgrind`**:

```
$ valgrind --tool=callgrind -- ./df103-fast
$ callgrind_annotate callgrind.out.<pid> | less
```

Would you want to run the full analysis with **`valgrind`**? How much longer did it take for the analysis to run under **`valgrind`** than without it? Where did the code spend most of its time?

5 Profiling with perf

Now, let's profile the same analysis with **`perf`**. Since the system version of **`perf`** was not built with support for the textual interface, we have installed **`perf-6.16`**, which has support for most **`perf`** features enabled. Please replace **`perf`** with **`perf-6.16`** if you want to use the `ncurses`-based text user interface (TUI).

```
$ perf record --freq 1000 -- ./df103-fast
$ perf report --stdio -s dso --percent-limit 1
$ perf report --stdio --no-children --percent-limit 1 --hierarchy
$ perf-6.16 record --all-cpus --call-graph=fp -e cycles:ukppp -- ./df103-fast
$ perf-6.16 report --stdio -g none --no-children --percent-limit 1
$ perf-6.16 report --inverted --sort comm
$ perf-6.16 record --call-graph=fp --latency -- ./df103-fast
$ perf-6.16 report --no-children --latency
$ perf-6.16 script | stackcollapse-perf.pl --addrs --jit --kernel | grep df103 > stacks.out
$ flamegraph.pl --bgcolors white --colors java < stacks.out > flamegraph.svg
```

Now use **`perf script`** to create a flamegraph of the analysis. What insights can you gain from it? For example, what percentage of CPU cycles was actually spent performing the analysis? What library takes most of CPU running time? Why? What would you change in this analysis to improve performance?

6 Profiling an Interpreted Analysis

You may have noticed that some frames in your flamegraph didn't look quite right. You probably also noticed that some symbols do not have their names correctly assigned. This is because ROOT's RDataFrame performs just-in-time (JIT) compilations before running the event loop. Although we have compiled the tutorial into a binary, some code was passed in to RDataFrame as strings, so these need to be converted to binary at runtime. The code that gets compiled during runtime, however, has no frame pointers and no debugging info by default, therefore the broken stack frames and missing symbol names. ROOT has an option to enable this, by setting `CLING_PROFILE=1` in the environment, which also enables writing symbol names and sizes to a **`perf`** map file, which **`perf`** can then use to assign proper names to JITted symbols. Let's recreate the flamegraph with this option enabled and compare with the previous one.

```
$ perf-6.16 record --call-graph=fp -e cycles:ppp -- env CLING_PROFILE=1 ./df103-fast
$ perf-6.16 report --no-children
$ perf-6.16 script --no-demangle | c++filt -p | stackcollapse-perf.pl --addrs --all > stacks.out
$ flamegraph.pl --bgcolors white --colors java < stacks.out > flamegraph.svg
```

Now you should be able to see JIT-compiled symbols with their proper names and in the right place in the flamegraph in light green (if using `colors=java` with **`flamegraph.pl`**).
Similarly, when running an analysis via ROOT's C++ interpreter, or with Python, this feature comes in handy to help resolve symbols that exist only at runtime. When used together with Python 3.12 and its support for **`perf`** with **`python3 -X perf`**, flamegraphs can also be constructed for interpreted analyses.

```
$ perf-6.16 record -g -- env CLING_PROFILE=1 root -l -q df103_NanoAODHiggsAnalysis.C
$ perf-6.16 record -g -- env CLING_PROFILE=1 python3 -X perf df103_NanoAODHiggsAnalysis.py
```

Create flamegraphs for the compiled, interpreted, and Python analysis scripts and compare them. What is the percentage of CPU cycles spent in the event loop (`ROOT::RDF::RunGraphs()`) in each case?

7 Running the Analysis on Local Data

Now, copy over the data from EOS onto the local disk (only the skimmed data, since the full dataset is 13G). Then, change the fast tutorial to use data from the local disk rather than from EOS.

```
$ xrdfs root://eospublic.cern.ch ls /eos/root-eos
$ xrdcp --recursive root://eospublic.cern.ch//eos/root-eos/cms_opendata_2012_nanoaod_skimmed .
$ ls -sh cms_opendata_2012_nanoaod_skimmed
$ sed -e '/eospublic/s@root://eospublic.cern.ch//eos/root-eos@.@' df103_NanoAODHiggsAnalysis_fast.C >
↪  df103_NanoAODHiggsAnalysis_local.C
$ $(root-config --cxx --cflags --libs) -O2 -g -o df103-fast-local df103_NanoAODHiggsAnalysis_local.C
```

Run the analysis using local data and compare with its remote counterpart. Is it faster when running locally?