

# L4: Experimental Design, Profiling, and Performance/Energy Optimization

---

M. Jam, P. de Oliveira Castro

September 29, 2025

Master Calcul Haute Performance et Simulation - GLHPC | UVSQ

1. Experimental Design, Profiling, and Performance/Energy Optimization

2. Experimental Methodology

3. Plotting Tools

4. Profiling

5. Live Demo

## Experimental Design, Profiling, and Performance/Energy Optimization

---

## Plot Example - Intro

In the following slides, you will be shown a series of plots; mainly taken from the PPN course reports of previous students.

For each plot:

- Try to understand what is represented
- Explain what you observe
- Give a **definitive** conclusion from the data shown

Raise your hands when ready to propose an explanation.

# Plot Example (1)

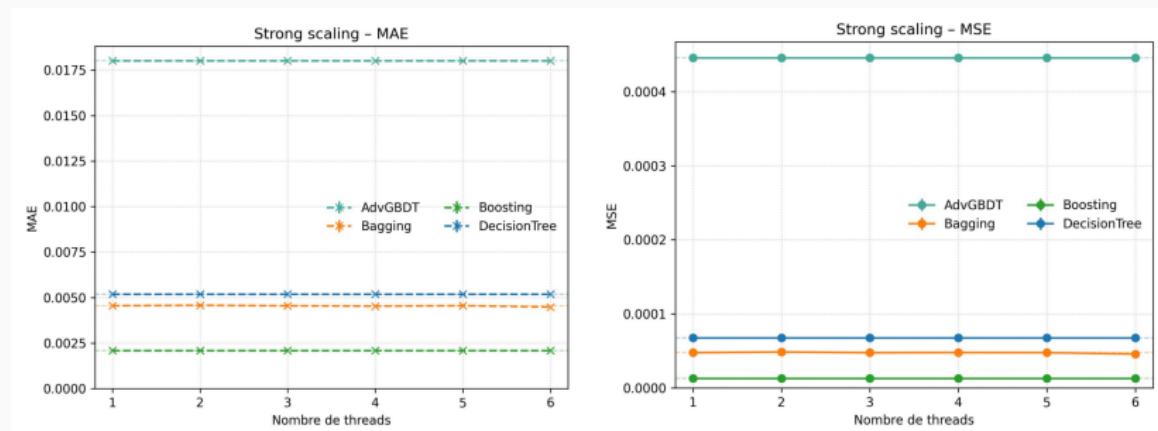


Figure 1: PPN Example - (No Caption)

## Plot Example (2)

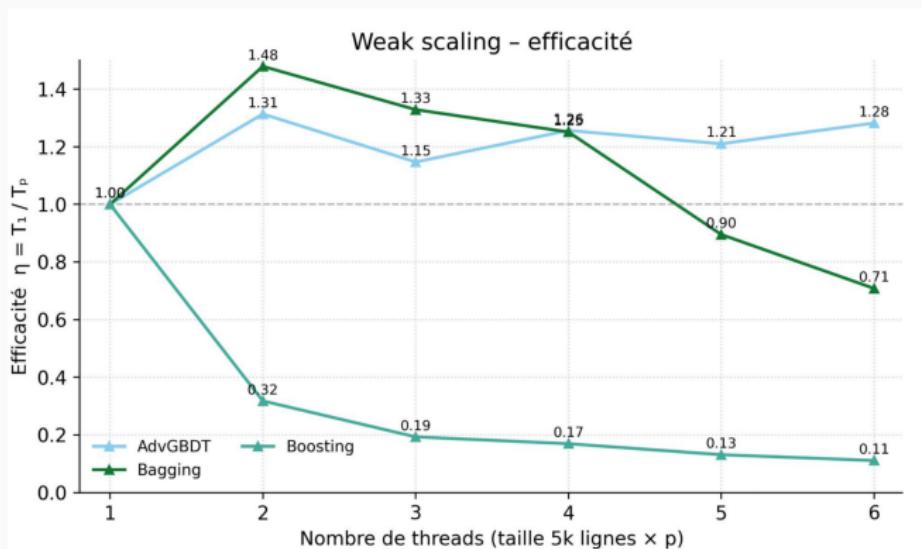


Figure 2: PPN Example - (No Caption)

## Plot Example (3)

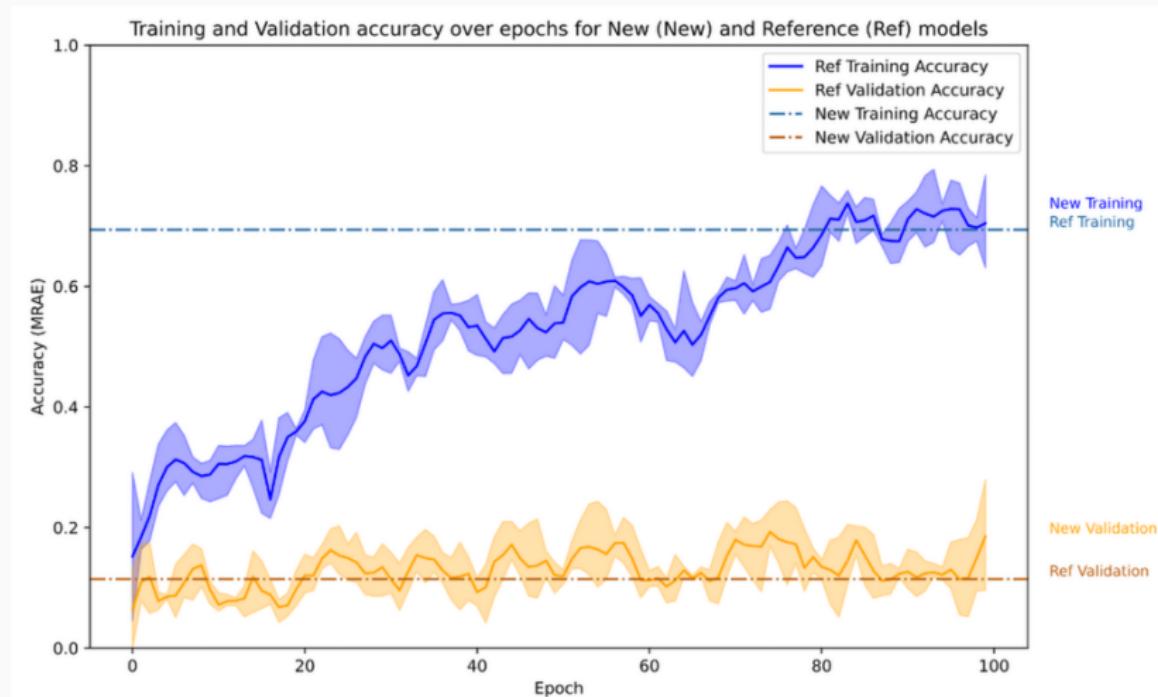


Figure 3: PPN Example - (No Caption)

## Plot Example (4)

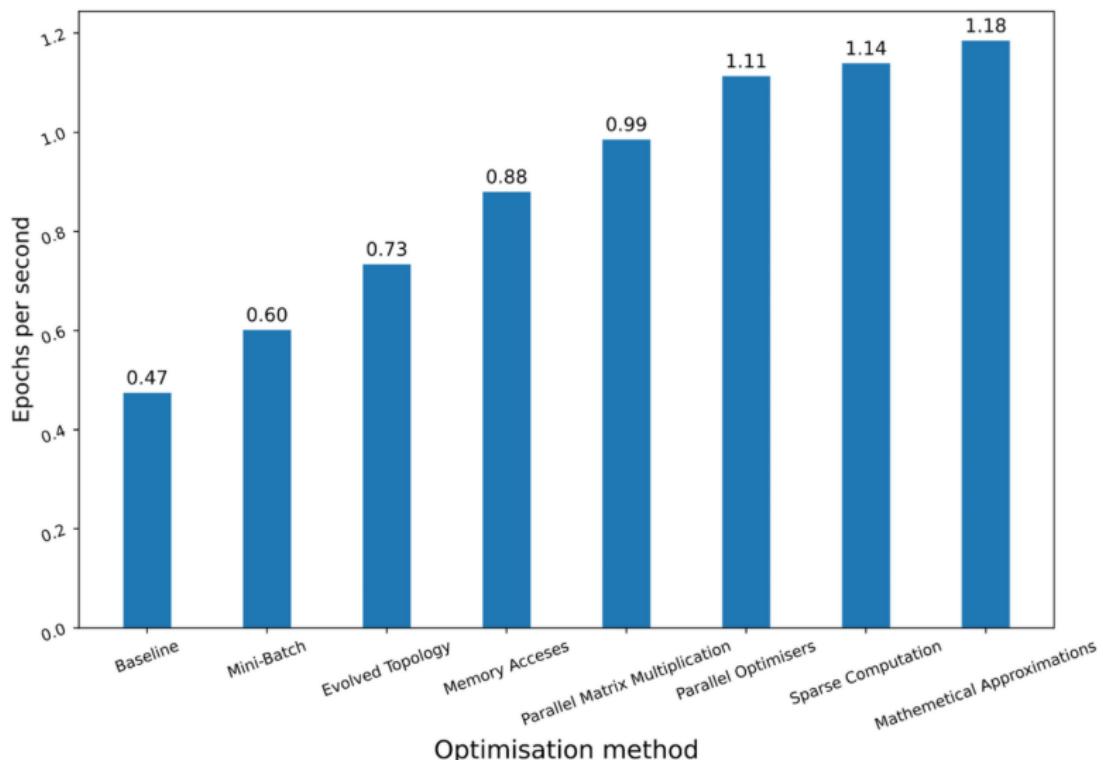


Figure 4: PPN Example - “Récapitulatif des optimisations faites”

## Plot Example (5)

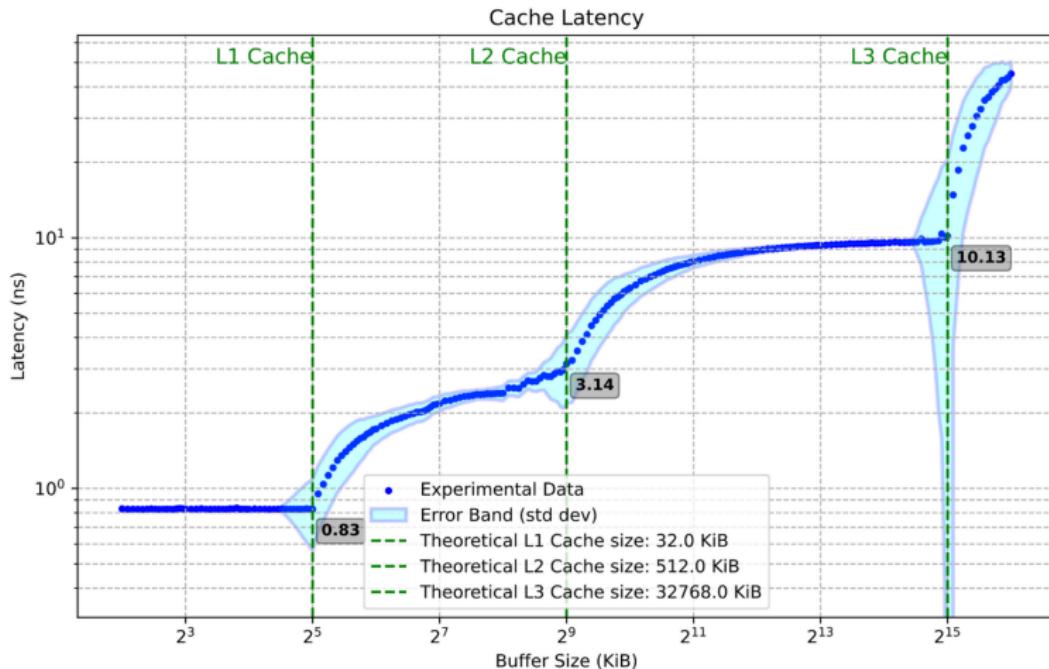


Figure 5: PPN Example - “Nouveau tracé de la latence cache”

## Plot Example (6)

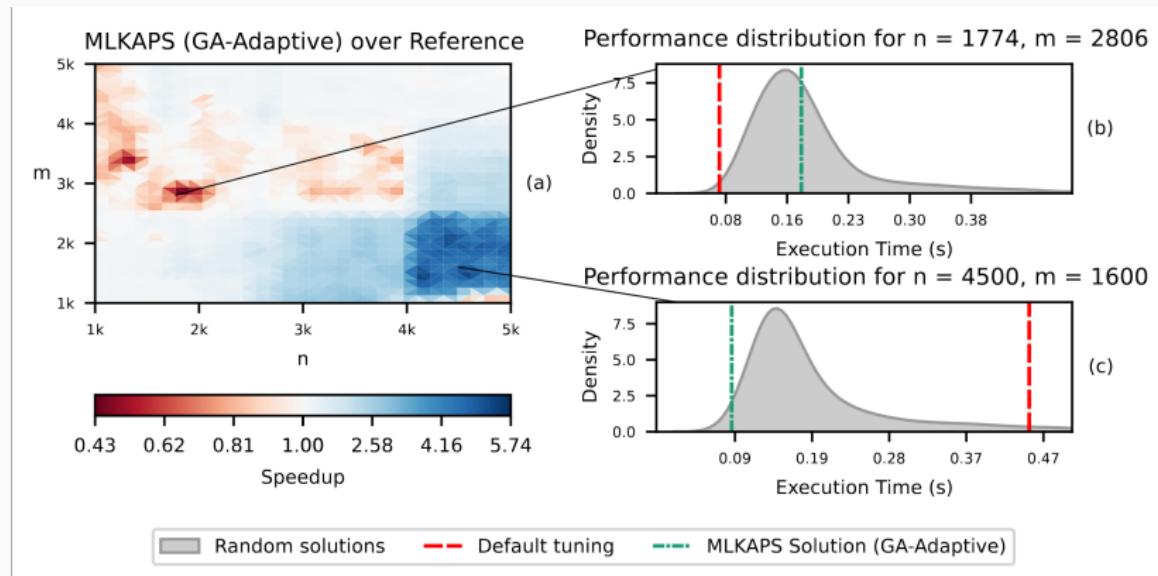
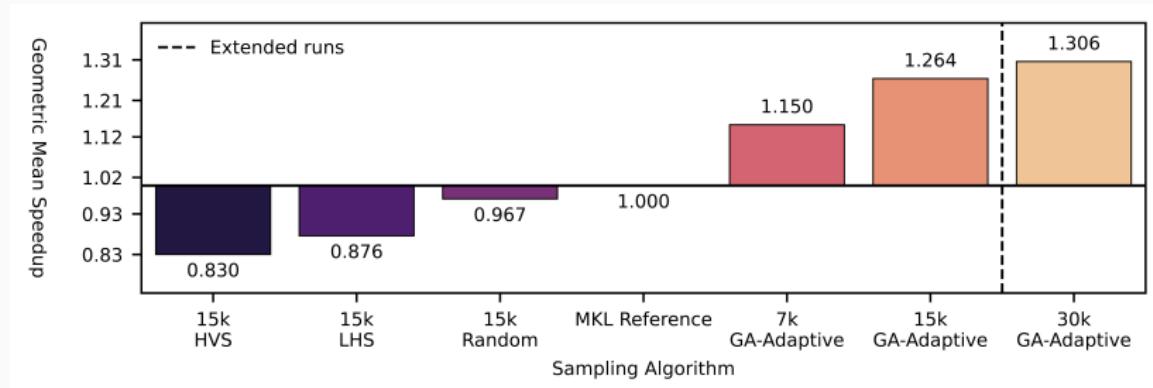


Figure 6: Prof Example - (KNM): (a) Speedup map of GA-Adaptive (7k samples) over the Intel MKL hand-tuning for `dgetrf` (LU), higher is better. (b) Analysis of the slowdown region (performance regression). (c) Analysis of the high speedup region. 3,000 random solutions were evaluated for each distribution.

## Plot Example (7)



**Figure 7: Prof Example - (SPR):** Geometric mean Speedup (higher is better) against the MKL reference configuration on `dgetrf` (LU), depending on the sampling algorithm. 46x46 validation grid. 7k/15k/30k denotes the samples count. GA-Adaptive outperforms all other sampling strategies for auto-tuning. With 30k samples it achieves a mean speedup of  $\times 1.3$  of the MKL `dgetrf` kernel.

## Plot Example - What makes a good plot

Ask yourself:

- What do I want to communicate ?
- What data do I need ?
- **Is my plot understandable in ~10 seconds ?**
- Is my plot self-contained ?
- Is the context, environment, and methodology clear ?

## Plot Example - Summary

HPC is a scientific endeavour; data analysis and plotting are essential.

- Plots drive decisions
- Plots make results trustworthy
- Plots explain complex behaviors

Datasets are large, multi-disciplinary, and often hard to reproduce.

## Experimental Methodology

---

# Experimental Methodology - Workflow



Figure 8: Typical experimental workflow

## Statistical significance - Introduction

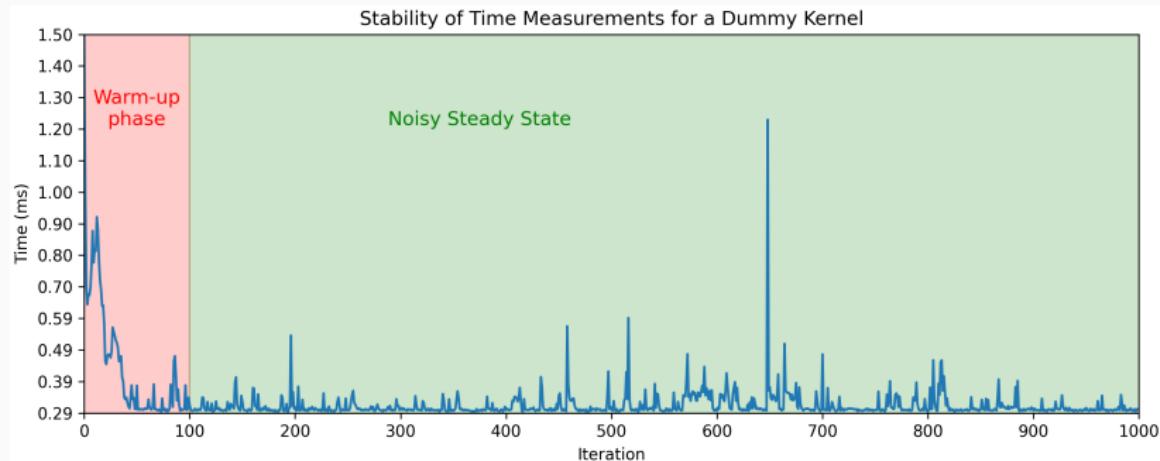
Computers are noisy, complex systems:

- Thread scheduling is non deterministic -> runtime varies between runs.
- Dynamic CPU frequency (Turbo/Boost)
- Systems are heterogeneous (CPU/GPU, dual socket, numa effects, E/P cores)
- Temperature/thermal throttling can alter runtime

How can we make sure our experimental measurements are reliable and conclusive?

## Statistical significance - Warm-up effects

Systems need time to reach steady-state:



On a laptop: Mean = 0.315 ms, CV = 13.55%

We need “warm-up” iterations to measure stable performance and skip cold caches, page faults, frequency scaling.

## Statistical significance - Noise mitigation

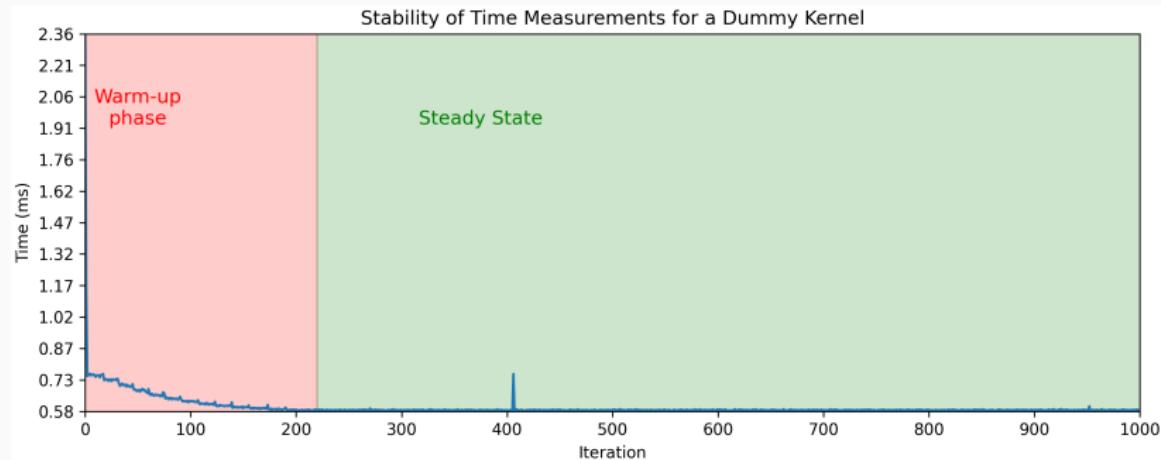
Noise can only be mitigated:

- Stop all other background processes (other users)
- Stabilize CPU Frequency (`sudo cpupower -g performance`)
  - Make sure laptops are plugged to avoid powersaving policies
- Pin threads via `taskset`, `OMP_PLACES` and `OMP_PROC_BIND`
- Consider hyperthreading
- Use stable compute nodes

Meta-repetitions are essential to mitigate noisy measurements.

# Statistical significance - Example

Same experiment on a stabilized benchmarking server:



On a laptop: Mean = 0.315 ms, CV = 13.55%

Stabilized node: Mean = 0.582 ms, CV = 1.14%

## Note

Timing on a laptop is always subpar

## Statistical significance - Mean, Median, Variance

Single-run measurements are misleading; we need statistics.

- Mean runtime  $\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$
- Median: less sensitive to outliers than the mean
- Variance/standard deviation: Measure of uncertainty
- Relative metrics are useful: Coefficient of variation  
 $(CV = \frac{\sigma}{\bar{x}} \times 100\%)$

We usually give both the mean and standard deviation when giving performance results. Plots usually show  $\bar{x} \pm 1\sigma$  as a shaded region around the mean to represent uncertainty.

### Note

Distribution plots can be useful: stable measurements are often close to Gaussian, even if systematic noise may lead to skewed or heavy-tailed distributions.

## Statistical significance - Confidence Intervals

How to decide how many repetitions we should perform ?

- Usually, the costlier the kernels, the less meta-repetitions are expected
- Short or really short kernels should have more metas to reduce the influence of noise

Remember that:

$$CI_{0.95} \approx \bar{x} \pm 1.96 \cdot \frac{\sigma}{\sqrt{n}}$$

More repetitions increase confidence, but returns diminish:

$$\text{CI width} \propto \frac{1}{\sqrt{n}}$$

### Note

Confidence intervals are a bit less common in plots than  $\pm 1\sigma$  but can also be used !

## Statistical significance - p-score & Hypothesis testing

In HPC, mean/median and variance often suffice, but hypothesis testing can become handy in some contexts.

- Null hypothesis ( $H_0$ ): GPU and CPU have the same performance for small matrixes
  - Differences in measurements are only due to noise
- Alternative hypothesis: CPU is faster for small matrixes
- p-value is the probability that  $H_0$  explains a phenomenon.
- If  $p < 0.05$ , we can safely reject  $H_0$  (Statistically significant difference)

Example:  $\bar{x}_{GPU} = 5.0\text{s}$ ,  $\sigma_{GPU} = 0.20$ ,  $\bar{x}_{CPU} = 4.8\text{s}$ ,  $\sigma_{CPU} = 0.4$ ,  
Two-sample t-test with 10 samples  $p = 0.02$ .

The measured differences between CPU and GPU execution time are statistically significant.

# Experimental Methodology – Reproducibility

Reproducibility is a very hot topic (Reproducibility crisis in science):

- Data and protocols are first-class citizens: as important as the plots themselves
- Transparency matters: make data, scripts, and parameters accessible
- Enables others to verify, build on, and trust your results

## Note

Beware of your mindset: your results should be credible and honest before being “good”.

“Our results are unstable, we have yet to understand why, this is what we tried” is a completely valid answer

## Plotting Tools

---

## Plotting tools - Cheetsheet

Name	Use
pandas	Storing and saving tabular data
numpy	Numerical arrays, manipulating data
matplotlib	Basic 2D plots, full control
seaborn	Statistical plots, higher-level API
logging	Logging experiment progress/results
OpenCV	Image processing, animations/videos
ffmpeg	Generating and encoding videos

Lookup the quick reference plotting gallery in the annex!  
Both `matplotlib` and `seaborn` provide extensive online galleries.

[Live Example of the `matplotlib` gallery  
<https://matplotlib.org/stable/gallery/index.html>]

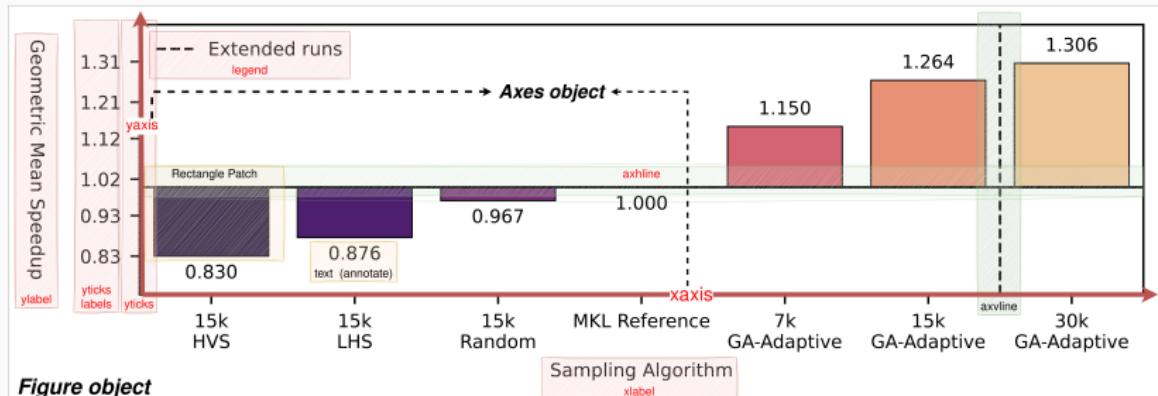
## Plotting tools - Matplotlib

Matplotlib is one of the most widely used plotting libraries.  
A figure is built hierarchically from nested elements:

- Figure (The canvas)
  - Subfigures
    - Axes (One or more subplots)
      - Axis (x/y/z scales, ticks, labels)
      - Artists (Lines, markers, text, patches, etc.)

- Data is plotted using axis-level functions like `ax.plot`,  
`ax.histogram`
- Customization occurs at both the Figure and Axes levels
- Complex multi plots layout occur at the Figure level

# Plotting tools - Matplotlib



## Plotting tools - Matplotlib

```
import matplotlib.pyplot as plt

x = [0, 1, 2, 3]
y = [2.8, 5.7, 12.5, 14]

# Create a new figure, single axis
# Size is 8 inches by 8 inches, and constrained layout
fig, ax = plt.subplots(figsize=(8, 8), layout="constrained")

# Plot a simple line
ax.plot(x, y, color="red", label="My Algorithm")

# Customize the axes
ax.set_xlabel("Iteration") # Name of the X axis
ax.set_ylabel("Time (s)") # Name of the y axis
# Title of the plot
ax.set_title("Evolution of Time with the number of iteration")

ax.margins(0, 0) # Remove white spaces around the figure
ax.legend(loc="upper right") # Draw the legend in the upper right
                             corner

fig.savefig("my_plot.png", dpi=300) # Higher DPI -> bigger image
plt.close() # End the plot and release resources
```

## Plotting tools - Matplotlib (Multi axis)

We can easily have multiple plots on the same figure:

```
nrows = 5, ncols = 1
fig, axs = plt.subplots(5, 1, figsize(8 * ncols, 3 * nrows))

ax = axs[0]
ax.plot()
...

ax = axs[1]
ax.plot()
...

fig.tight_layout() # Alternative to constrained layout
fig.savefig("my_muliplot.png", dpi=300)
```

Each axis is its own plot, with its own legend and artists.

### Note

Use the reference (<https://matplotlib.org/stable/api/index.html>) and gallery (<https://matplotlib.org/stable/gallery/index.html>) extensively !

## Plotting tools - Seaborn

Seaborn is an extension of Matplotlib dedicated to statistical visualization:

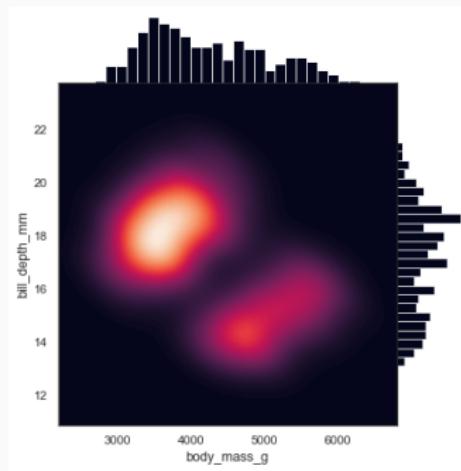


Figure 9: <https://seaborn.pydata.org/examples/index.html>

It's useful for histograms, bar charts, kdeplots, scatterplots, and is overall a very good companion library.

## Plotting tools - Seaborn

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np

df = pd.read_csv(...) # Read the dataframe from somewhere

fig, ax = plt.subplots(figsize=(8, 8), layout="constrained")

# We must pass the axis to plot on as an argument
sns.kdeplot(data=df, x="Time", label="Algorithm", color="red",
    fill=True, ax=ax)

ax.set_title("Distribution of Execution time for the algorithm")
ax.margins(0, 0)
ax.set_xlabel("Time (s)", fontweight="bold")
ax.set_ylabel("Density", fontweight="bold")

ax.set_xticks(np.linspace(df["Time"].min(), df["Time"].max(), 10)
# Format the x axis ticks: `3.25s`
ax.xaxis.set_major_formatter(StrMethodFormatter("{x:.2f}s"))

fig.savefig("my_distribution.png")
```

## Profiling

---

## Profiling - Motivation

- HPC codes are massive, complex and heterogeneous
- Humans are **bad** at predicting bottlenecks
- Don't blindly optimize everything
- Profiling guides optimization

Remember: Always profile first.

## Profiling - Amdahl's law

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{S}}$$

Where  $f$  is the fraction of program improved, and  $S$  is the speedup on that fraction.

Example:

- I have optimized 80% of my application, with a speedup of x10
- In total, my application is now  $\frac{1}{0.2+(0.8/10)} = 3.57 \times$  faster

The 20% are a bottleneck !

# Profiling - Steps

## 1. Where (Hotspots) ?

- What functions are we spending time/energy in ?
- What call-tree are we spending time/energy in ?

## 2. Why ?

- Arithmetic density, memory access patterns
- Cache misses, branch misspredictions, vectorization efficiency  
(Hardware counters)

## 3. What goal ?

- Should I optimize for speed ? For energy ? Memory footprint ?
  - What about cold storage size/compression ?
- Do I have constraints (i.e. limited memory) ?
- Should I optimize or switch algorithm ?

## Profiling - Time

It's rather easy to benchmark a single function using a (high-resolution monotonic) clock:

```
begin = time.now()  
my_function()  
end = time.now()  
elapsed = end - begin
```

Very simple way to evaluate a function cost

## Profiling - Time (Stability)

But we have to account for noise:

```
for _ in range(NWarmup):
    my_function()

times = []
for _ in range(NMeta):
    begin = time.perf_counter()
    my_function()
    times.append(time.perf_counter() - begin)

median = np.median(times)
std = np.std(times)
print(f"Time: {median} +/- {std}")
```

We must check that our measures are valid !

## Profilers - Introduction

Full application -> Thousands of functions to measure !

- Profilers are tools to automate this
- Two main types:
  - Sampling: Pause the program and log where the program is  
(Costly functions -> More samples !)
  - Instrumentation: Modify the program to automatically add timers

Profilers can also check for thread usage, vectorization, memory access, etc.

# Perf - Record

Linux Perf is a powerful and versatile profiler:

```
perf record -g -- python3 ./scripts/run_bls.py kepler-8
[ perf record: Woken up 255 times to write data ]
[ perf record: Wrote 85.474 MB perf.data (1220354 samples) ]

perf report perf.out
```

Samples: 696K of event 'cpu_core/cycles/Pu', Event count (approx.): 651792089127			
Children	Self	Command	Shared Object
-	98.66%	98.65%	python3
-	98.61%	bls.so	bls._omp_fn.0
+	0.96%	0.96%	python3
	0.35%	0.00%	libscipy_openblas-68440149.so
	0.35%	0.00%	libc.so.6
	0.35%	0.00%	python3
	0.35%	0.00%	libpython3.13.so.1.0
	0.35%	0.00%	python3
	0.34%	0.00%	libpython3.13.so.1.0
	0.31%	0.06%	libpython3.13.so.1.0
	0.31%	0.00%	libpython3.13.so.1.0
	0.30%	0.00%	python3
	0.30%	0.00%	libpython3.13.so.1.0
			[.] bls._omp_fn.0
			[.] blas_thread_server
			[.] __libc_start_call_main
			[.] __libc_start_main@@GLIBC_2.34
			[.] Py_BytesMain
			[.] _start
			[.] Py_RunMain
			[.] _PyEval_EvalFrameDefault
			[.] PyImport_ImportModuleLevelObject
			[.] object_vacall
			[.] PyObject_CallMethodObjArgs
			[.] cfunction_vectorcall_FASTCALL_KEYWORDS
			[.] PyEval_EvalCode
			[.] builtin_exec

It's a great tool to quickly get a Tree stack without any dependencies.

## Profiling - Hardware counters

In reality, perf is not really a profiler !

- The Linux Perf API can be used to access many hardware counters
- Perf record is just one usage of perf

Most CPUs/GPUs have hardware counters that monitor different events:

- Number of cycles
- Number of instructions
- Number of memory access
- RAPL

## Profiling - Perf for Hardware counters

```
perf stat -e cycles,instructions python3 ./scripts/run_bls.py  
kepler-8  
{'period': 3.520136229965055, 'duration': 0.11662673569985234, '  
phase': 0.43, 'depth': 0.004983530583444806, 'power':  
0.028861678651344452}  
  
Performance counter stats for 'python3 ./scripts/run_bls.py  
kepler-8':  
  
 962,187,248,452      cpu_atom/cycles/  
                        (43.66%)  
 1,119,319,677,606    cpu_core/cycles/  
                        (56.34%)  
 3,547,146,665,075    cpu_atom/instructions/  
                        3.69  insn per cycle      (43.66%)  
 2,837,633,772,530    cpu_core/instructions/  
                        2.54  insn per cycle      (56.34%)  
  
                         12.507192456 seconds time elapsed
```

The Intel VTune profiler is more complex but more self-contained than perf:

**Elapsed Time**: 13.630s

- CPU Time: 296.280s
- Total Thread Count: 55
- Paused Time: 0s

**Top Hotspots**

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
blas_omp_fn_0	blis.so	292.060s	98.6%
blas_thread_server	libscipy_openblas-68440149.so	2.900s	1.0%
_lfc_start_main	lfc.so.6	0.059s	0.3%
memset	libc-dynamic.so	0.140s	0.6%
operator new	libc++abi.so	0.088s	0.0%
[Others]	N/A*	0.233s	0.3%

\*N/A is applied to non-summable metrics.

**Hotspots Insights**

If you see significant hotspots in the Top Hotspots list, switch to the [Bottom-up](#) view for in-depth analysis per function. Otherwise, use the [Caller/callee](#) or the [Frame Graph](#) view to track critical paths for these hotspots.

**Explore Additional Insights**

Parallelism: 77.6%

Use [Threaded](#) to explore more opportunities to increase parallelism in your application.

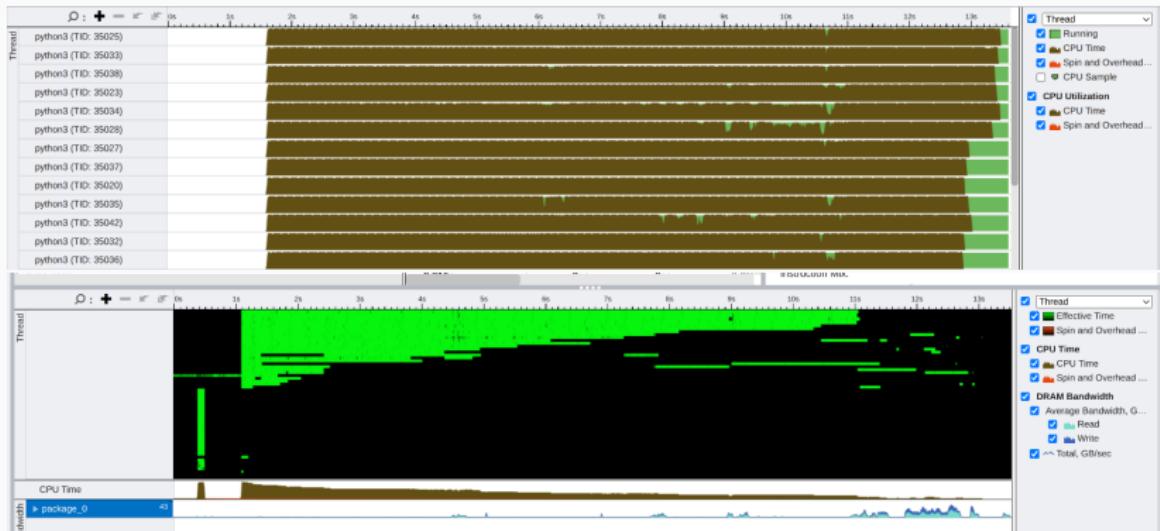
Vectorization: 0.0%

Use [HPC Performance Characterization](#) to learn more on vectorization efficiency of your application. A significant fraction of floating point arithmetic instructions are scalar. Use [Intel Advisor](#) to see possible reasons why the code was not vectorized.

The figure displays two histograms side-by-side. The left histogram shows 'Elapsed Time' on the y-axis (0 to 8s) against 'Poor' on the x-axis (0 to 20). It has a single bar at index 1 labeled 'Idle'. The right histogram shows 'Average Effective CPU Utilization' on the y-axis (0 to 16%) against 'Logical CPUs' on the x-axis (0 to 28). It has a large bar at index 27 labeled 'Idle'.

38/46

# VTune - CPU Usage



# VTune - HPC Performance

VTune has multiple collection mode:

## Memory Bound 1.0% of Pipeline Slots

### Performance-core (P-core):

Memory Bound  1.0% of Pipeline Slots

### Efficient-core (E-core):

Memory Bound  0.3% of Clockticks

\*N/A is applied to metrics with undefined value. There is no data to calculate the metric.

### Bandwidth Utilization Histogram

## Vectorization 0.0% of Packed FP Operations

### Instruction Mix:

SP FLOPs  0.0% of uOps

Packed  0.0% from SP FP

Scalar  100.0%  from SP FP

DP FLOPs  16.1% of uOps

Packed  0.0% from DP FP

Scalar  100.0%  from DP FP

x87 FLOPs  0.0% of uOps

Non-FP  83.9% of uOps

Metrics were collected from Big Cores only.

## Top Loops/Functions with FPU Usage by CPU Time

This section provides information for the most time consuming loops/functions with floating point operations.

Function	CPU Time  % of FP Ops 	FP Ops: Packed 	FP Ops: Scalar 	Vector Instruction Set 	Loop Type 
[Loop@0x1940 in bls_.omp_fn.0]	120.926s 16.3%	0.0%	100.0% 	SSE2(128) 	
[Loop@0x184f in bls_.omp_fn.0]	0.106s 3.2%	0.0%	100.0%		
native_write_msr	0.040s 1.6%	0.0%	100.0%		
[Loop@0x1920 in bls_.omp_fn.0]	0.029s 10.5%	0.0%	100.0%	SSE(128); SSE2(128)	

\*N/A is applied to non-summable metrics.

## Other profilers

- MAQAO is a profiler developed by the LIPARAD
- AMD, NVIDIA and ARM have their own profilers for their platforms
- And many, many others (likwid, gprof, etc.)

Usually, we combine a “quick” profiler like gprof/perf record with a more indepth one when needed.

## Profiling - Energy

Energy is a growing concern:

- One HPC cluster consume millions of dollars in electricity yearly
- ChatGPT and other LLM are computationally intensive:
  - Nvidia GPUs consumes lots of energy

On the flip side, measuring energy is harder than measuring time.

Many actors still focus on execution time only -> Energy is perceived as "Second rank"

## Profiling - RAPL

Running Average Power Limit (RAPL) is an x86 hardware counter that monitors energy consumption:

- Energy is tracked at different level
  - Core, Ram, Package, GPU, etc.
- It does not account for secondary power consumers (Fans, Water cooling, etc.)
- RAPL is not event based: The entire machine is measured ! (Background processes, etc.)

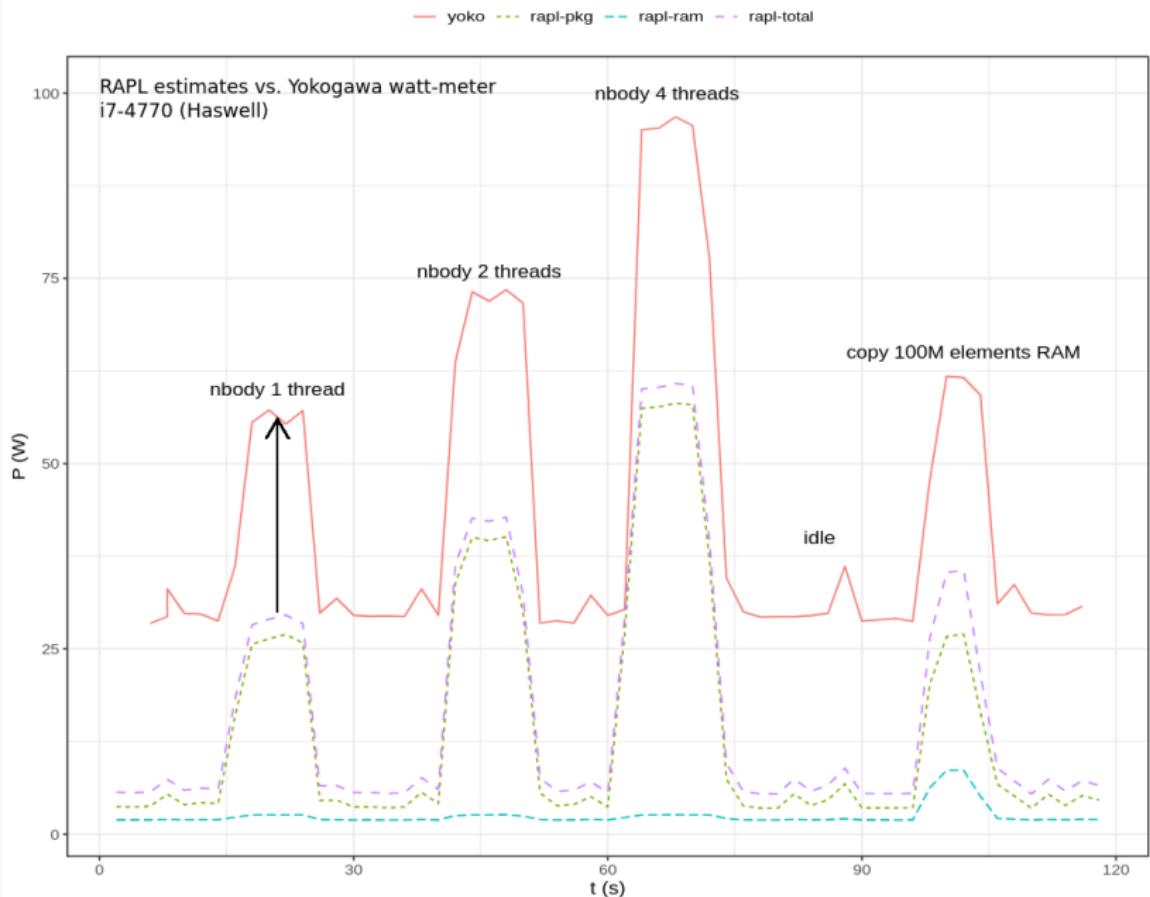
It requires sudo permissions to access (compared to a clock)

```
perf stat -a -j -e power/energy-pkg,power/energy-cores <app>
{"counter-value" : "88.445740", "unit" : "Joules", "event" : "power/energy-pkg/", "event-runtime" : 10002168423, "pcnt-running" : 100.00}
{"counter-value" : "10.848633", "unit" : "Joules", "event" : "power/energy-cores/", "event-runtime" : 10002166697, "pcnt-running" : 100.00}
```

# Profiling - Watt-Meter



# Profiling - RAPL accuracy



## Live Demo

---

## Experiment example

Annex/run\_experiment.sh

Annex/model\_convergence.py