Teratec Hackathon. Edition#2.

"La Team"

2024-01-29

Acknowledgments

We extend our sincere gratitude to the Teratec Hackathon organizers for providing us with the opportunity to participate in this inspiring event. Special thanks to the entire organizing team for their dedication and support throughout the competition.

Introduction

From Monday January 22 to Friday January 26, 2024, we took part in the hackathon organized by Teratec. We had the opportunity to work on two problems (which we will present in this report) provided by CGG and EDF. The HPC platform made available is based on AWS Graviton 3 processors, fueled by Arm Neoverse technologies.

This report will enable us to explain everything we have understood, done and tried, to show the results we were able to achieve and explain what we would have done with more time on our hands. It will also be the opportunity to present how we worked, what our organization was during this week, and so much more.

We'll also give a more personal conclusion on what this hackathon has brought us, both pedagogically and in terms of teamwork.

”La Team”

We are four students in the second year of the CHPS Master's degree at the University of Reims Champagne-Ardenne.

During this hackathon, we organized ourselves rather simply within the team. We set up a GitHub repository to version our code correctly. Every half-day we took stock of our tasks. We started by splitting into two groups to learn about the two challenges.

You can see our work by following this link!

Contents

Figure 1: The Migouélou Dam in the Pyrenees (France / EDF courtesy)

1 Telemac

In response to the challenge presented by the Hackathon, our team enthusiastically embarked on the task of adapting the Telemac software for utilization on an ARM cluster. Telemac, a robust suite developed by the National Institute for Industrial Environment and Risks (INERIS) in France, holds a central role in the realm of environmental and hydraulic engineering. This Fortran-based software, specifically designed for High-Performance Computing (HPC) environments, employs MPI (Message Passing Interface) for efficient parallel processing. Telemac's versatility shines in its ability to simulate intricate hydrodynamic processes, making it a go-to tool for understanding river flows, tidal patterns, and sediment transport.

The significance of Telemac extends to its diverse modules, providing researchers and engineers with a comprehensive platform to model and analyze various aquatic environments. Covering river dynamics, estuarine processes, and coastal interactions, Telemac's capabilities align seamlessly with the challenges of water-related phenomena.

In this report, we dive into the details of our collaborative efforts to seamlessly port Telemac onto an ARM cluster. Our focus spans the intricacies of modification, compilation, runtime execution, benchmarking, and validation. Through this thorough exploration, we aim to illuminate the specific adaptations made, the challenges encountered, and the successful outcomes achieved. This exhaustive endeavor underscores our commitment to pushing the boundaries of software optimization in contemporary HPC environments.

1.1 Gfortran install

Utilizing the GCC compiler, I compiled the Telemac suite for ARM by following the standard installation process outlined on the Telemac Wiki. This approach, guided by the community's documentation, allowed for a straightforward integration onto the ARM architecture, providing a reliable foundation for subsequent benchmarking and validation steps on the cluster.

1.2 Armflang install

This section describes how to port the Telemac software to the ARM architecture using an ACFL compiler. The compiler used is ACFL/23.10 and the OpenMPI version is 4.1.6.

1.2.1 Pysource config

Changing the pysource config is the first thing to do.Although it looks similar to the previous config file, we can see that it points to a different config systel file and that a new environment variable has been added to the path. This variable will be explained later.

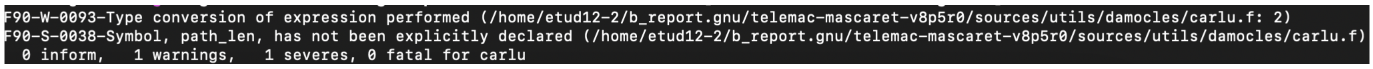
1.2.2 Systel config

The systel file is the file that configures compilation, it sets up which compiler to use and which flags to use. It is also almost identical to the previous one. In this file we didn't change the gfortran calls, because the [gfortran] configuration is not called. Finally, in the [acflfortranHPC] configuration, the compiler is still MPIF90, but as mentioned in the previous part, it's ACFL's MPIF90. One thing to note is that we have added a flag to the MPIF90 flags. This flag will be relevant later to fix some problems we encounter when compiling.

1.2.3 Compiling

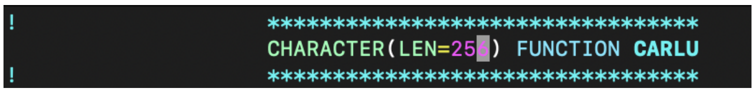
In this part, we will review all the errors we have encountered when compiling the Telemac software with the ACFL compiler and how to fix them.

After compiling the Telemac software with the vanilla file, i.e. the same file from [gfortranHPC], but with MPIF90 from ACFL. The first error we encounter is the following :



```
F90-W-0093-Type conversion of expression performed (/home/etud12-2/b_report.gnu/telemac-mascaret-v8p5r0/sources/utils/damocles/carlu.f: 2)
F90-S-0038-Symbol, path_len, has not been explicitly declared (/home/etud12-2/b_report.gnu/telemac-mascaret-v8p5r0/sources/utils/damocles/carlu.f)
  0 inform,   1 warnings,   1 severes, 0 fatal for carlu
```

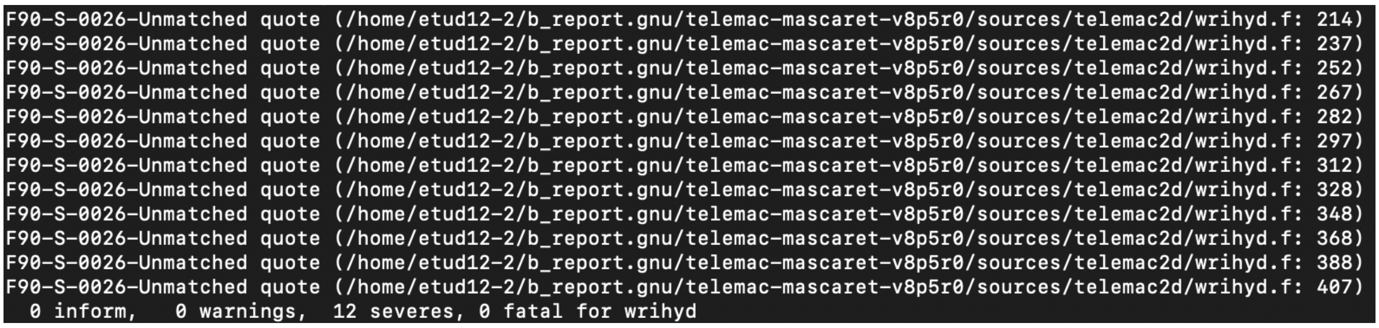
Figure 2: path_len error in "carlu.f"



```
!                 *********************************
                  CHARACTER(LEN=256) FUNCTION CARLU
!                 *********************************
```

Figure 3: path_len fix in "carlu.f"

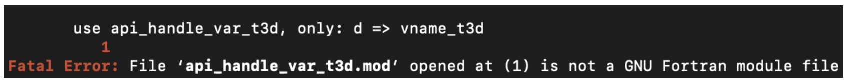
This error is in the file "carlu.f", we fixed it by hardcoding the length of the character table to 256. We also found this error in "mycarlu.f", which we also fixed. Then the next error encountered is in the file "wrihyd.f"



```
F90-S-0026-Unmatched quote (/home/etud12-2/b_report.gnu/telemac-mascaret-v8p5r0/sources/telemac2d/wrihyd.f: 214)
F90-S-0026-Unmatched quote (/home/etud12-2/b_report.gnu/telemac-mascaret-v8p5r0/sources/telemac2d/wrihyd.f: 237)
F90-S-0026-Unmatched quote (/home/etud12-2/b_report.gnu/telemac-mascaret-v8p5r0/sources/telemac2d/wrihyd.f: 252)
F90-S-0026-Unmatched quote (/home/etud12-2/b_report.gnu/telemac-mascaret-v8p5r0/sources/telemac2d/wrihyd.f: 267)
F90-S-0026-Unmatched quote (/home/etud12-2/b_report.gnu/telemac-mascaret-v8p5r0/sources/telemac2d/wrihyd.f: 282)
F90-S-0026-Unmatched quote (/home/etud12-2/b_report.gnu/telemac-mascaret-v8p5r0/sources/telemac2d/wrihyd.f: 297)
F90-S-0026-Unmatched quote (/home/etud12-2/b_report.gnu/telemac-mascaret-v8p5r0/sources/telemac2d/wrihyd.f: 312)
F90-S-0026-Unmatched quote (/home/etud12-2/b_report.gnu/telemac-mascaret-v8p5r0/sources/telemac2d/wrihyd.f: 328)
F90-S-0026-Unmatched quote (/home/etud12-2/b_report.gnu/telemac-mascaret-v8p5r0/sources/telemac2d/wrihyd.f: 348)
F90-S-0026-Unmatched quote (/home/etud12-2/b_report.gnu/telemac-mascaret-v8p5r0/sources/telemac2d/wrihyd.f: 368)
F90-S-0026-Unmatched quote (/home/etud12-2/b_report.gnu/telemac-mascaret-v8p5r0/sources/telemac2d/wrihyd.f: 388)
F90-S-0026-Unmatched quote (/home/etud12-2/b_report.gnu/telemac-mascaret-v8p5r0/sources/telemac2d/wrihyd.f: 407)
  0 inform,   0 warnings,  12 severes, 0 fatal for wrihyd
```

Figure 4: Unmatched quote error in "wrihyd.f"

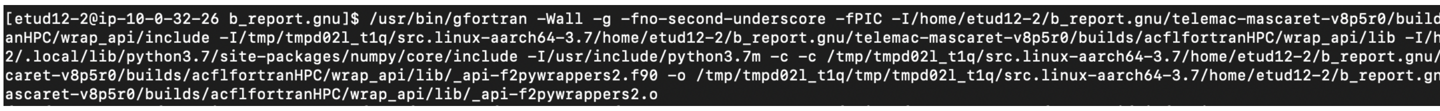
You may not get this error if you have copied our systel file. In fact, this error has been fixed by the flag mentioned in the systel config part, the "-Mbackslash" flag, which tells the compiler to read the file in C style.
At this point there is only one error remaining in the whole compilation, which in our opinion was the most difficult to fix:



```
      use api_handle_var_t3d, only: d => vname_t3d
         1
Fatal Error: File ‘api_handle_var_t3d.mod’ opened at (1) is not a GNU Fortran module file
```

Figure 5: .mod not readable by GNU compiler error

Again, you will not get this error if you have copied our pysource file. This is where the environment variable becomes relevant ! Well, let's delve further into the error to see why. Actually, this error comes from the f2py3 configuration, when invoked it calls the f-compiler which is gfortran, so it can't open the mod files. We can see this by adding the flag "–verbose" to "compile_telemac.py" :

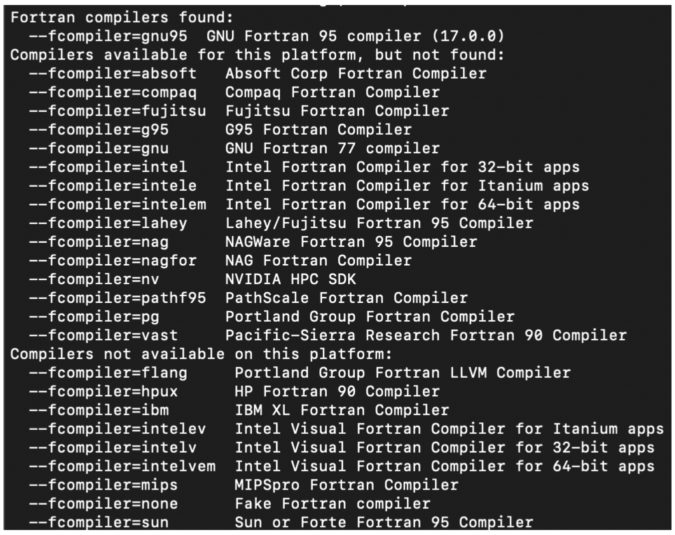


```
[etud12-2@ip-10-0-32-26 b_report.gnu]$ /usr/bin/gfortran -Wall -g -fno-second-underscore -fPIC -I/home/etud12-2/b_report.gnu/telemac-mascaret-v8p5r0/build
anHPC/wrap_api/include -I/tmp/tmpd02l_t1q/src.linux-aarch64-3.7/home/etud12-2/b_report.gnu/telemac-mascaret-v8p5r0/builds/acflfortranHPC/wrap_api/lib -I/h
2/.local/lib/python3.7/site-packages/numpy/core/include -I/usr/include/python3.7m -c -c /tmp/tmpd02l_t1q/src.linux-aarch64-3.7/home/etud12-2/b_report.gnu/
caret-v8p5r0/builds/acflfortranHPC/wrap_api/lib/_api-f2pywrappers2.f90 -o /tmp/tmpd02l_t1q/tmp/tmpd02l_t1q/src.linux-aarch64-3.7/home/etud12-2/b_report.gn
ascaret-v8p5r0/builds/acflfortranHPC/wrap_api/lib/_api-f2pywrappers2.o
```

Figure 6: The command line invoked by f2py3

As mentioned above, f2py3 calls the gfortran compiler. To fix this, we first think about the system config file, if we change the [pyd_fcompiler] from "gnu95" to "armflang" it would work. Unfortunately, the f2py3 f-compiler does not recognise armflang as one of its parameters. Worse still, none of its supported parameters recognised a module compiled by armflang, so we were at a dead end.

To find a way to solve this problem, we first tried compiling the same line without going through the wrapper with the right compiler, i.e. armflang. Once we were sure it would work with the right compiler we just had to find a way to give it to f2py3, we tried to pass it with the "–fcompiler-exec" flag to pass an absolute path to the armflang compiler, unfortunately this flag was already deprecated. Then, by analysing the compilation options and the compilation logs, we noticed that /usr/bin/gfortran was the base compiler in the F90-compiler flag. That's when we had the idea of exporting the environment variable to modify this flag, which relies on it, thus we were able to finalise the compilation of Telemac with armflang.



```
Fortran compilers found:
  --fcompiler=gnu95  GNU Fortran 95 compiler (17.0.0)
Compilers available for this platform, but not found:
  --fcompiler=absoft   Absoft Corp Fortran Compiler
  --fcompiler=compaq   Compaq Fortran Compiler
  --fcompiler=fujitsu  Fujitsu Fortran Compiler
  --fcompiler=g95      G95 Fortran Compiler
  --fcompiler=gnu      GNU Fortran 77 compiler
  --fcompiler=intel    Intel Fortran Compiler for 32-bit apps
  --fcompiler=intele   Intel Fortran Compiler for Itanium apps
  --fcompiler=intelem  Intel Fortran Compiler for 64-bit apps
  --fcompiler=lahey    Lahey/Fujitsu Fortran 95 Compiler
  --fcompiler=nag      NAGWare Fortran 95 Compiler
  --fcompiler=nagfor   NAG Fortran Compiler
  --fcompiler=nv       NVIDIA HPC SDK
  --fcompiler=pathf95  PathScale Fortran Compiler
  --fcompiler=pg       Portland Group Fortran Compiler
  --fcompiler=vast     Pacific-Sierra Research Fortran 90 Compiler
Compilers not available on this platform:
  --fcompiler=flang     Portland Group Fortran LLVM Compiler
  --fcompiler=hpux      HP Fortran 90 Compiler
  --fcompiler=ibm       IBM XL Fortran Compiler
  --fcompiler=intelev   Intel Visual Fortran Compiler for Itanium apps
  --fcompiler=intelv    Intel Visual Fortran Compiler for 32-bit apps
  --fcompiler=intelvem  Intel Visual Fortran Compiler for 64-bit apps
  --fcompiler=mips      MIPSpro Fortran Compiler
  --fcompiler=none      Fake Fortran compiler
  --fcompiler=sun       Sun or Forte Fortran 95 Compiler
```

Figure 7: List of all fcompiler available in f2py3

1.3 Run Telemac using Slurm

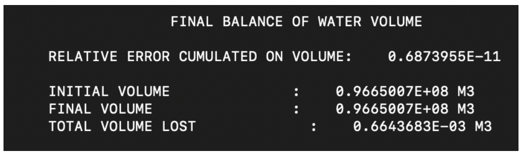
In configuring the use of Slurm, we examined the configuration files employed at EDF to discern the means of instructing Telemac to submit MPI jobs through Slurm. We made subtle modifications to the configuration, resulting in the settings provided below. These settings are integrated into either the GNU or ACFL configuration, enabling the execution of Telemac jobs on the cluster.

```
# EDF workaround
sbatch_tag:#SBATCH
hpc_stdin: #!/bin/bash
   [sbatch_tag] --job-name=<jobname>
   [sbatch_tag] --output=<jobname>-<time>.out
   [sbatch_tag] --error=<jobname>-<time>.err
   [sbatch_tag] --time=<walltime>
   [sbatch_tag] --ntasks=<ncsize>
   [sbatch_tag] --exclusive
   [sbatch_tag] --nodes=<ncnode>
   [sbatch_tag] --ntasks-per-node=<nctile>
   source <root>/../pysource.<configName>.sh
   <py_runcode>
#
mpi_cmdexec: map --profile mpirun -np <ncsize> <exename>
hpc_runcode: bash <root>/scripts/submit_slurm.sh <id_log> '<project>'
par_cmd_exec: srun -n 1 -N 1 <config>/partel < <partel.par> >> <partel.log>
```

1.3.1 Validation

For validation, due to lack of time, we only used the codes indicated in the instructions. In this way, all our codes run to completion whether they are executed from the armflang wrapper or the gfortran wrapper. All the programmes we have been asked to check are run to completion.

Some of them turned out to be wrong, in particular due to division by 0, because during the runs the Telemac version did not have the MED library. Here are some of the output files we obtained during the various executions :

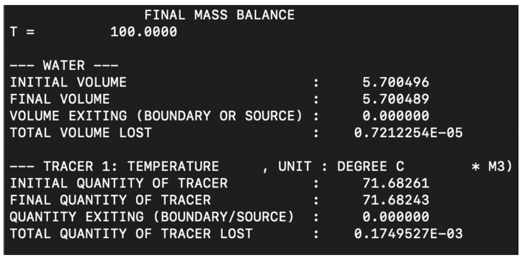


```
                   FINAL BALANCE OF WATER VOLUME

     RELATIVE ERROR CUMULATED ON VOLUME:    0.6873955E-11

     INITIAL VOLUME              :    0.9665007E+08 M3
     FINAL VOLUME                :    0.9665007E+08 M3
     TOTAL VOLUME LOST             :    0.6643683E-03 M3
```

Figure 8: Run of Telemac2D ”malpasset.fine” on 6 nodes and 384 tasks.



```
                FINAL MASS BALANCE
T =          100.0000

--- WATER ---
INITIAL VOLUME                      :     5.700496
FINAL VOLUME                        :     5.700489
VOLUME EXITING (BOUNDARY OR SOURCE) :     0.000000
TOTAL VOLUME LOST                   :    0.7212254E-05

--- TRACER 1: TEMPERATURE     , UNIT : DEGREE C        * M3)
INITIAL QUANTITY OF TRACER          :     71.68261
FINAL QUANTITY OF TRACER            :     71.68243
QUANTITY EXITING (BOUNDARY/SOURCE)  :     0.000000
TOTAL QUANTITY OF TRACER LOST       :    0.1749527E-03
```

Figure 9: Run of Telemac3D ”bump_static” on 4 nodes and 64 tasks.

1.3.2 Scalability

Like the previous part, this part was carried out in the final rush. The observations made on these codes and all the screenshots come from compilation with the gfortran wrapper, as we didn’t have time to implement MAP on the armflang wrapper.

2 The Zeros of Riemann

Figure 10: Our Dear Georg Friedrich Bernhard Riemann.

Presentation of the problem

We were tasked with optimizing a serial and inefficient implementation of a code that calculates the number of zeros of the Riemann **Zeta** function along the critical line within a specified imaginary part interval. The original code employed the Riemann-Siegel formula but lacked the necessary optimizations for performance. Our team took on the challenge of enhancing the code's efficiency by implementing strategic optimizations and transformations. By addressing bottlenecks and improving the overall computational approach, we aimed to significantly reduce execution time.

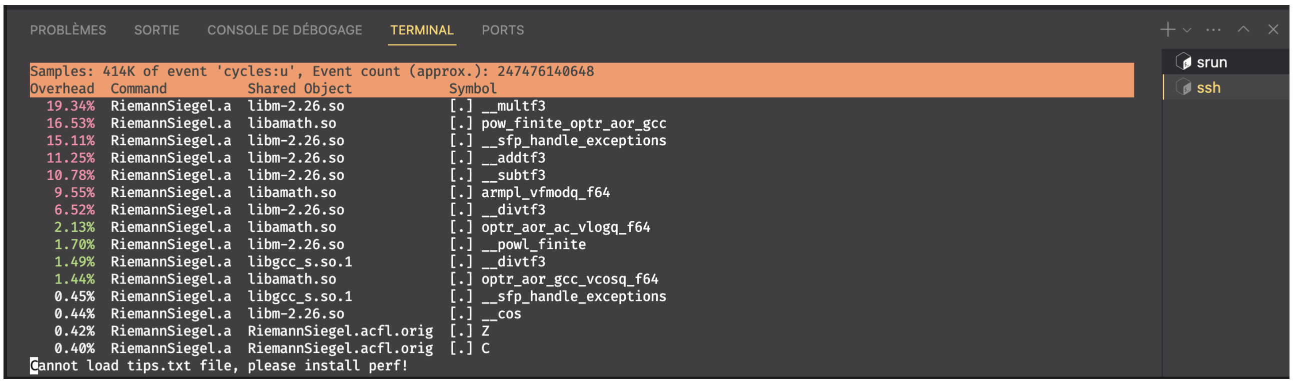
Our aim was to find a sweet spot between accuracy and speed, making the code more efficient overall.

2.1 Statistics with the baseline code and first comments

We quickly realised that the ACFL compiler was better at optimizing the code to the given ARM architecture. Our later benchmarks were thus based on using *armclang*.[2]

We explored optimal compilation options for our code. Naturally, we opted for maximum optimizations using the compiler flag `-O3`. Additionally, we instructed the compiler to target the specific ARM Graviton v3 processors with the `-mcpu=neoverse-v1` option[1]. To harness the full potential of our ARM architecture, we leveraged the ARM Performance Libraries (ARM PL) provided by the Hackathon organization team, specifying the `-armpl` flag[3].

Using the test case $(10.0, 1000000.0, 100.0)$ we evaluated the efficiency and cornered the bottlenecks of the original code.



```
Samples: 414K of event 'cycles:u', Event count (approx.): 247476140648
Overhead  Command           Shared Object             Symbol
  19.34%  RiemannSiegel.a   libm-2.26.so              [.] __multf3
  16.53%  RiemannSiegel.a   libamath.so               [.] pow_finite_optr_aor_gcc
  15.11%  RiemannSiegel.a   libm-2.26.so              [.] __sfp_handle_exceptions
  11.25%  RiemannSiegel.a   libm-2.26.so              [.] __addtf3
  10.78%  RiemannSiegel.a   libm-2.26.so              [.] __subtf3
   9.55%  RiemannSiegel.a   libamath.so               [.] armpl_vfmodq_f64
   6.52%  RiemannSiegel.a   libm-2.26.so              [.] __divtf3
   2.13%  RiemannSiegel.a   libamath.so               [.] optr_aor_ac_vlogq_f64
   1.70%  RiemannSiegel.a   libm-2.26.so              [.] __powl_finite
   1.49%  RiemannSiegel.a   libgcc_s.so.1             [.] __divtf3
   1.44%  RiemannSiegel.a   libamath.so               [.] optr_aor_gcc_vcosq_f64
   0.45%  RiemannSiegel.a   libgcc_s.so.1             [.] __sfp_handle_exceptions
   0.44%  RiemannSiegel.a   libm-2.26.so              [.] __cos
   0.42%  RiemannSiegel.a   RiemannSiegel.acfl.orig   [.] Z
   0.40%  RiemannSiegel.a   RiemannSiegel.acfl.orig   [.] C
Cannot load tips.txt file, please install perf!
```

Figure 11: Perf report of ./RiemannSiegel 10 100000 100

2.2 Serial code optimizations

The main hotspots of the code are the **Z** and **C** functions computation.

The original code spends a lot of its computation time in mathematical functions, such as **logs**, **cos**, **powers**, **square-roots**. Our aim was first to find workarounds the use of these functions even if it means sacrificing some precision over speed.

We refined the code with this goal in mind, reported the achieved speedups, and identified persistent bottlenecks in the process.

- `inline`-revision

  Math constants are computed at compile-time using `constexpr`
  Functions `Z`, `C`, `theta` and `even` are inlined using the `static inline` keywords.

- `theta`-revision

  The expression of `theta` is of the form:

$$\begin{aligned}\theta(t) &= \frac{t}{2}\log\frac{t}{2\pi} - \frac{t}{2} + a_0 + \frac{a_1}{t} + \frac{a_3}{t^3} + \frac{a_3}{t^3} + \frac{a_5}{t^5} + \frac{a_7}{t^7} + \frac{a_9}{t^9} \\ &= \frac{t}{2}\log\frac{t}{2\pi} + P(\frac{1}{t})\end{aligned}$$

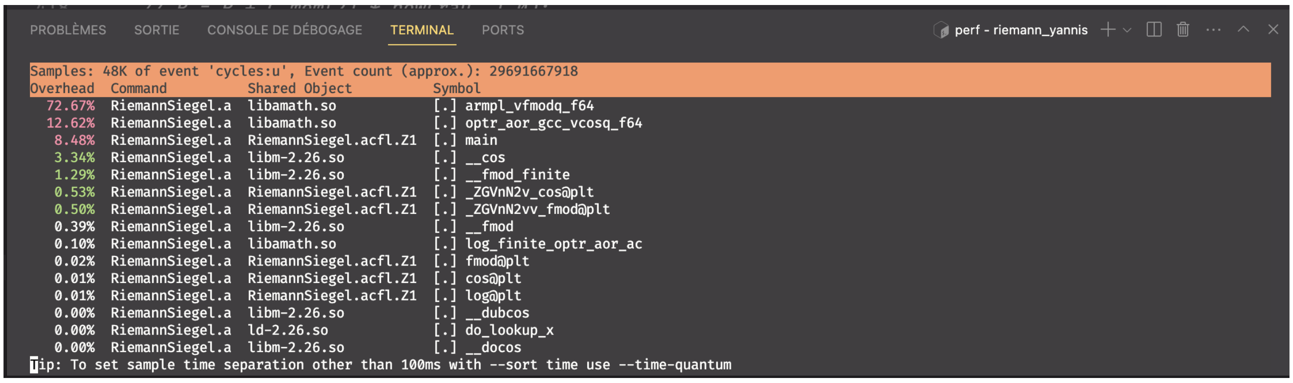
  With P being polynomial, we can thus rewrite the polynomial evaluation part of `theta` computation using the Horner scheme[4]. Furthermore the ARM architecture allow every operation of type `acc = a + b * acc` to use only one floating-point operation, enabling even more performance.

- `C`-revision

Every coefficient computation calculated by `C` is polynomial. We can use the same optimization as `C`-theta. We also removed `(int) n` from being a parameter of the function and instead `C` computes and returns all the coefficients at the same time as they are all used by `Z`.

- `Z1`-revision
  - Math constants replacements.
  - N_{max} computation along with $\log(j)$ and $\frac{1}{\sqrt{j}}$ memoization for all $1 \leq j \leq N_{max}$
  - Half-integer power conversion to power of square-roots.
  - `R` polynomial expression is computed using **Horner scheme**.
- `Z2`-revision

At this point, the biggest hotspots of the `Z` computation [Figure 12] are the `fmod` and `cos` functions.



```
Samples: 48K of event 'cycles:u', Event count (approx.): 29691667918
Overhead  Command           Shared Object            Symbol
  72.67%  RiemannSiegel.a   libamath.so              [.] armpl_vfmodq_f64
  12.62%  RiemannSiegel.a   libamath.so              [.] optr_aor_gcc_vcosq_f64
   8.48%  RiemannSiegel.a   RiemannSiegel.acfl.Z1    [.] main
   3.34%  RiemannSiegel.a   libm-2.26.so             [.] __cos
   1.29%  RiemannSiegel.a   libm-2.26.so             [.] __fmod_finite
   0.53%  RiemannSiegel.a   RiemannSiegel.acfl.Z1    [.] _ZGVnN2v_cos@plt
   0.50%  RiemannSiegel.a   RiemannSiegel.acfl.Z1    [.] _ZGVnN2vv_fmod@plt
   0.39%  RiemannSiegel.a   libm-2.26.so             [.] __fmod
   0.10%  RiemannSiegel.a   libamath.so              [.] log_finite_optr_aor_ac
   0.02%  RiemannSiegel.a   RiemannSiegel.acfl.Z1    [.] fmod@plt
   0.01%  RiemannSiegel.a   RiemannSiegel.acfl.Z1    [.] cos@plt
   0.01%  RiemannSiegel.a   RiemannSiegel.acfl.Z1    [.] log@plt
   0.00%  RiemannSiegel.a   libm-2.26.so             [.] __dubcos
   0.00%  RiemannSiegel.a   ld-2.26.so               [.] do_lookup_x
   0.00%  RiemannSiegel.a   libm-2.26.so             [.] __docos
Tip: To set sample time separation other than 100ms with --sort time use --time-quantum
```

Figure 12: Overhead generated by `fmod` before the optimization

The `fmod` function, which calculates the remainder of a double divided by another, was causing significant overhead when called repeatedly during the execution of the Zeta function. Despite optimizations with armclang++, the overhead remained high. As a solution, we identified an alternative method that sacrifices some floating-point precision in exchange for faster computation of the remainder in double division[5].

```
double x, fmod_x;

x = tt - t * MEM_LOG_J[j]; // tt is theta(t)
// fmod_x = fmod(x, TWO_PI);
fmod_x = fma(trunc(x / TWO_PI), -TWO_PI, x);
```

`fma` is a mathematical function that executes a fused multiply-add operation and efficiently truncates the fractional part of a double. This functionality aids in computing the remainder of a double division. As a result of this optimization, the execution time is halved, albeit with a minor loss in precision.

- `Z3`-revision

  The latest identified bottleneck in the computation of Z is attributed to the *cosinus* function. To enhance efficiency, we aim to calculate $\cos(x)$ more effectively, where x falls within the range $[-2\pi, 2\pi]$. Exploiting the even nature of the *cosinus*, our focus narrows to $cos(|x|)$. Let α represent $|x|$. Utilizing trigonometric identities, we derive:

$$\begin{aligned}\cos(\alpha) &= -\cos(\alpha - \pi) \\ &= -\cos(|\alpha - \pi|) \\ &= -\cos(\beta) \quad (\beta = |\alpha - \pi| \in [0, \pi]) \\ &= -\sin\left(\frac{\pi}{2} - \beta\right) \\ &= -\sin(\beta^*) \quad (\beta^* = \frac{\pi}{2} - \beta \in [0, \frac{\pi}{2}])\end{aligned}$$

  Computing $\cos(\alpha)$ is tantamount, up to a sign reversal, to computing $\sin(\beta)$ with β^* ranging from 0 to $\frac{\pi}{2}$. Consequently, we employ a Taylor series approximation[6]:

$$\sin(\beta^*) \approx \beta^* - \frac{(\beta^*)^3}{3!} + \frac{(\beta^*)^5}{5!} - \frac{(\beta^*)^7}{7!} + \frac{(\beta^*)^9}{9!} + \dots$$

  Through empirical testing, we determined that using powers up to the 9th was sufficient to ensure the required precision for our computation. To optimize the polynomial evaluation, we employed the Horner Scheme, pre-computing coefficients as constants in the code. This optimization resulted in the computation time of the *cosinus* being reduced by half.

In conclusion, the amalgamation of techniques such as inlining, polynomial reformulation, the Horner scheme, and Taylor expansion led to a substantial **37.35**-fold overall speedup on the ARM architecture of the serial code. The optimization journey is summarized in the table below.

Revision	**Result**	**Time (s)**	**Speedups (/n-1)**
orig	138069	95.745	NaN
inline	138069	86.408	1.11
theta	138069	20.93	4.13
C	138069	14.514	1.44
Z1	138069	11.429	1.27
Z2	138069	4.945	2.31
Z3	138069	2.55	1.94

Table 1: Comparison of `./RiemannSiegel-<revision> 10 100000 100`

2.3 Parallelization (w/ OpenMP)

Our priority in this problem was not necessarily to parallelize the code, but to understand the function behind it and optimize it mathematically. However, there's no denying that parallelization is a non-negligible gain in performance.

We therefore decided to parallelize the main loop, using OpenMP. We considered a number of approaches, including task-based ones, but settled on a basic solution consisting of simply determining a batch number according to the number of threads available (64). This approach is not yet complete (but that's the point of a hackathon! to come up with ideas, and test them if time permits).

```
int num_chunks = omp_get_max_threads();
int chunk_size = 1 + (NUMSAMPLES + num_chunks - 1) / num_chunks;
```

We believe that a dynamic approach with more accurate chunk computation could make parallelization even more efficient.

We also wanted to add an if clause to determine the number of threads in relation to the size of the problem. However, even with small problems, we noticed that the maximum number of threads was not penalizing. After all, a large number of operations are performed in the Z function anyway.

2.4 Adaptative sampling and Convergence

This part is more about the research we've done into the convergence of the function and its behavior in general with large problems. For that we simply traced the results for the problem 100000. You can see the result in the following figure:

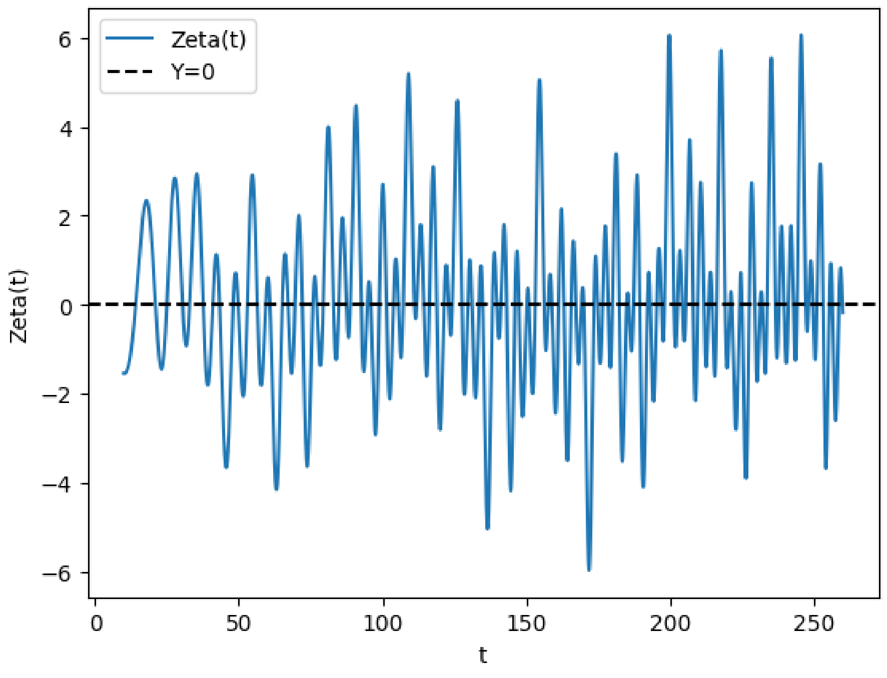


Figure 13: Compute of the trace of Zeta Function (10 100000 100)

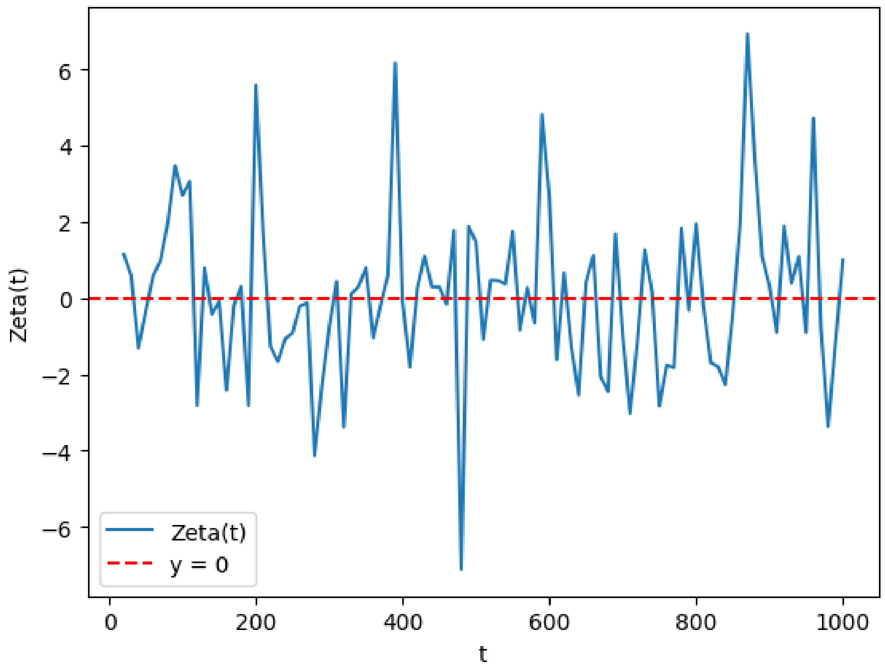


Figure 14: Compute of the trace of Zeta Function (10 100000 0.1)

Using a bigger sample step we obtain the following figure:

In the figure above, we observed that consecutive points sometimes obscure potential zeroes. For instance, when three consecutive positive points have the middle one smaller than either of its neighbors, it indicates the presence of a missed zero. In such cases, the interval should be resampled with a finer granularity. The same rationale applies to negative points.

This suggests the feasibility of initiating sampling with a very small value, subsequently increasing it in regions of interest. Eventually, with a finer granularity, this approach reduces the computational load for zero counting.

While we haven't implemented this solution due to time constraints, we recognize its potential to efficiently address substantial problems.

Our approach would have entailed commencing with modest sampling values and progressively scaling up to attain precise zero counting. This would have been accomplished through our combination of serial optimization and parallelization to optimize the computational efficiency of finer granularity searches.

2.5 Benchmark

This benchmark section will allow us to present all the results we have obtained. There are several points we'd like to present: the general speedup, the scalability study, the resolution of larger problems, but also the comparison between the different compilers used.

exec	compiler	revision	result	time (s)
RiemannSiegel.acfl.orig	acfl	orig	138069	95.240
RiemannSiegel.acfl.inline	acfl	inline	138069	86.415
RiemannSiegel.acfl.theta	acfl	theta	138069	20.930
RiemannSiegel.acfl.C	acfl	C	138069	14.511
RiemannSiegel.acfl.Z1	acfl	Z1	138069	11.435
RiemannSiegel.acfl.Z2	acfl	Z2	138069	4.945
RiemannSiegel.acfl.Z3	acfl	Z3	138069	2.550
RiemannSiegel.acfl.ZX	acfl	ZX	138069	0.054

Table 2: Final comparison for the problem 100000 (step = 100)

2.5.1 Speedup

2.5.2 Scalability (weak vs strong)

As we mentioned earlier, we had better performance with armclang++ than with g++. However, the mathematical optimizations we were able to code had a huge impact on g++ performance. The following graph shows the comparison between the two, following each problem.

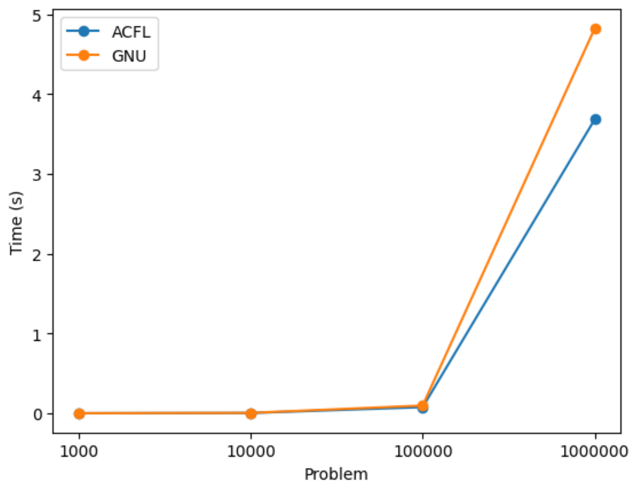


Figure 15: Comparison of g++ and armclang++ in the first problems

It's also interesting to compare the two compilers on each version of our optimizations. On the 8 different codes, we have calculated the average speed up obtained. The speedups are only computed for problems between 1000 and 100000.

As you can see on the figure 16, the **ZX** code does not appear in the speedup. In fact, it's much higher than those on the graph: we reach a maximum speedup of **386666668266.95483** with gnu and **276666668050.08887** with acfl. But the average speedup obtained with just sequential optimisations is already really impressive.

In the fig**??**, the number of threads is in fact the exponent (so 1, 2, 4, 8, 16, 32 and 64 threads in total, so the maximum in one node).

With the help of certain optimizations, we realized that we could converge faster with a reduced number of steps. In fact, this is not yet the deeper study of the function we'd like to

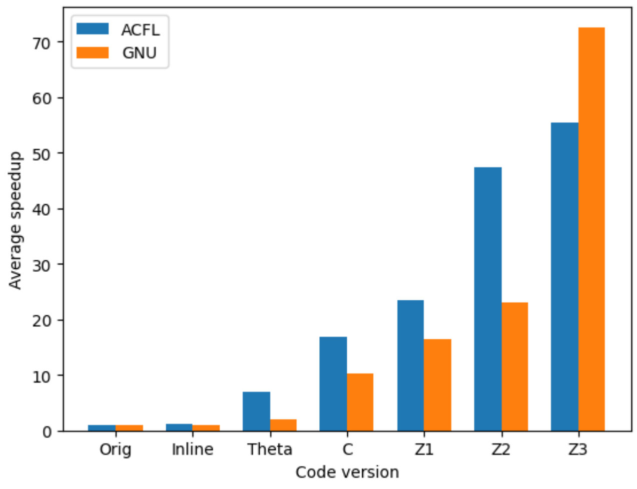
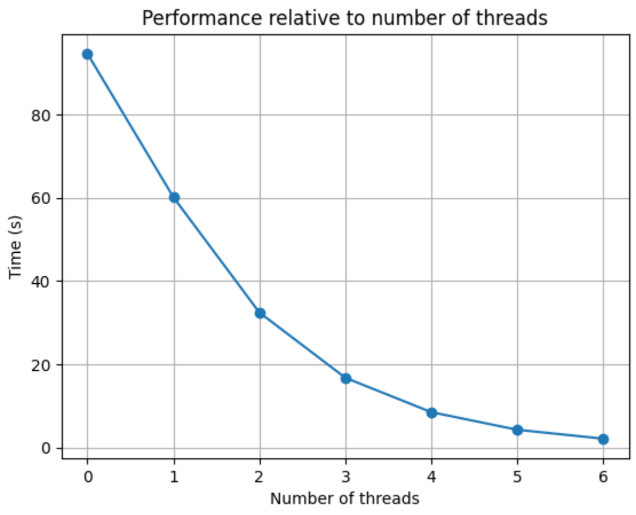


Figure 16: Average speedup obtained on weak scaling



have made, but it comes very close. In the table that follows, we show the minimum time it took us to solve a problem, with the maximum number of steps required.

Keep in mind in this section that the number of 0's found is correct, even with the approximations we've built into the code. This is not necessarily true for larger problems.

2.6 Conclusion

In concluding this segment on the Riemann-Siegel approach, although low-level optimizations and parallelization have facilitated a substantial speedup (approximately 1760), we recognize that dedicating time to implement more sophisticated mathematical transformations and our adaptive sampling grain strategy could have further enhanced the performance of the code. This, in turn, would have enabled the computation of even larger intervals.

Altogether, this phase of the Hackathon presented significant challenges, prompting us to generate creative solutions and develop a deeper understanding of the problem to enhance the efficiency of the code.

Problem	**nb Steps**	**time (s)**
1000	10	0.000 (!)
10000	20	0.001
100000	32	0.017
1000000	171	2.318

Table 3: Number of steps before convergence

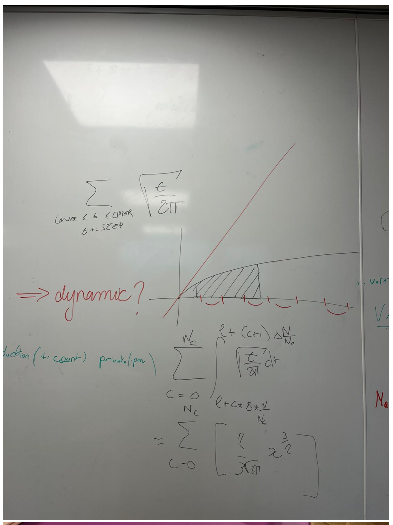
3 General conclusions on the Hackathon

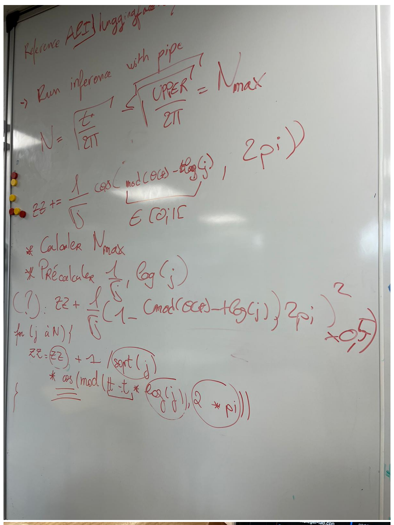
In conclusion, our participation in the Teratec, EDF, and CGG-organized Hackathon provided an enriching experience that seamlessly merged theoretical knowledge gained during our Master's degree in High-Performance Computing (HPC) with practical applications in real-world challenges. The dual nature of the exercise, involving porting and testing Telemac on an ARM cluster and optimizing a C++ code for calculating the zeroes of the Zeta function using the Riemann-Siegel formula, allowed us to delve deeply into both hardware and software optimization domains.

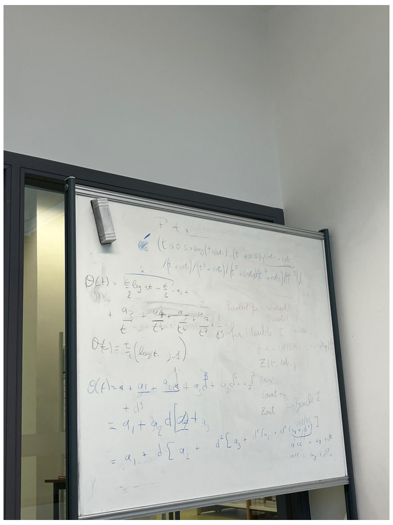
Throughout the Hackathon, we faced not only technical hurdles but also organizational challenges, particularly in terms of workload distribution. Addressing these challenges provided us with a valuable lesson in teamwork and collaboration. We strategically leveraged our individual strengths, ensuring that each team member played to their expertise, thus minimizing overall weaknesses through effective work sharing. This experience not only showcased the versatility of our collective skills but also highlighted the significance of effective project management in the realm of high-performance computing.

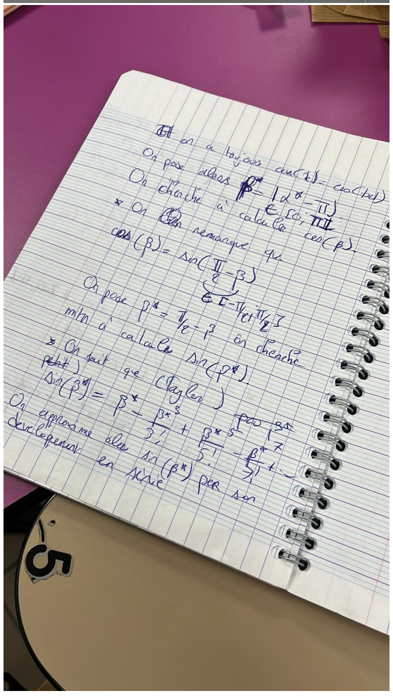
In summary, the Hackathon not only allowed us to apply our theoretical knowledge in a practical setting but also equipped us with invaluable insights into the intricacies of teamwork and project management. This experience served as a testament to the multifaceted nature of HPC, where technical prowess and collaborative skills intertwine to overcome challenges and deliver impactful solutions. We express our gratitude to Teratec and its sponsors for organizing this event, providing a platform for us to push the boundaries of our capabilities and contribute meaningfully to the field of high-performance computing.

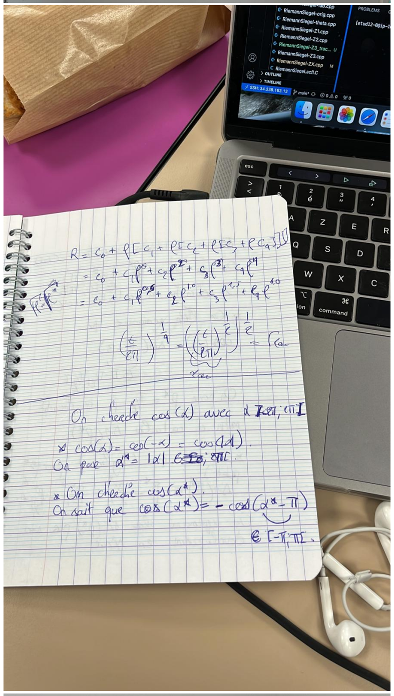
In conclusion, we'd like to share some snapshots from the board and notes we compiled during this Hackathon. :)

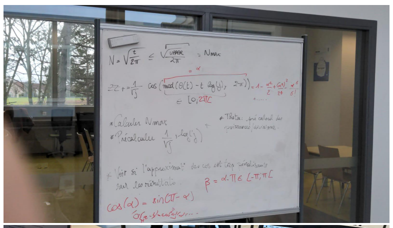

=> dynamic?

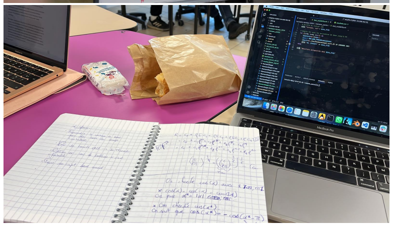

Reference AES lugg
-> Run inference with pipe
N = sqrt(t / 2π) ≤ sqrt(UPPER / 2π) = Nmax
* Calculer Nmax
* Précalculer 1/sqrt(j), log(j)

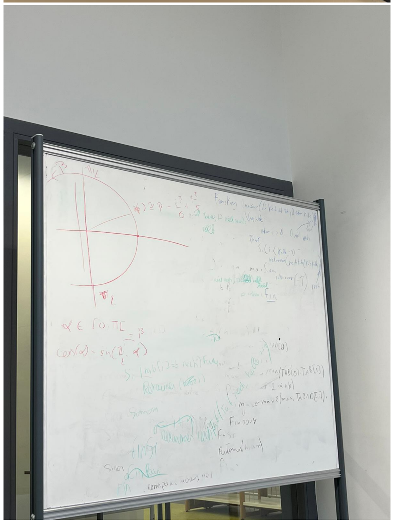


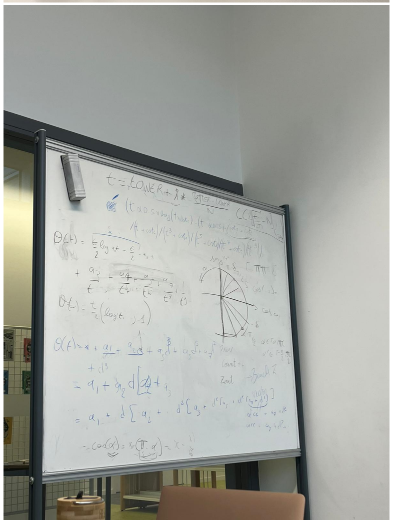

On a toujours cos(α) = cos(|α|)
On pose alors β* = |α* − π/2|
On cherche à calculer cos(β)
On remarque que cos(β) = sin(π/2 − β)
On pose β* = π/2 − β on cherche
m/n à calculer sin(β*)
On sait que (Taylor)
On applique alors le développement en série


On cherche cos(α) avec α ∈ [−π, π]
cos(α) = cos(−α) = cos(|α|)
On pose α* = |α| ∈ [0, π]
On cherche cos(α*)
On sait que cos(α*) = −cos(α* − π)
∈ [−π, π]









References

[1] *Arm Compiler for Linux: what is new in the 22.0 release?* 2022. URL: https://community.arm.com/arm-community-blogs/b/tools-software-ides-blog/posts/arm-compiler-for-linux-and-arm-performance-libraries-22-0.

[2] *Choosing Compilers for HPC on Arm*. 2023. URL: https://community.arm.com/arm-community-blogs/b/high-performance-computing-blog/posts/choosing-compilers-for-arm-hpc.

[3] *Get started with Arm Performance Libraries in Arm Compiler for Linux*. 2023. URL: https://developer.arm.com/documentation/102574/2310/?lang=en.

[4] *Horner's method*. 2024. URL: https://en.wikipedia.org/wiki/Horner%27s_method.

[5] *Implementation of FMOD function*. 2020. URL: https://stackoverflow.com/questions/26342823/implementation-of-fmod-function.

[6] *Taylor's Series of sin x*. 2024. URL: https://ocw.mit.edu/courses/18-01sc-single-variable-calculus-fall-2010/242ad6a22b86b20799afc7f207cd4271_MIT18_01SCF10_Ses99c.pdf.