**What did you learn from Vtune profiling:**

I’ve used Vtune Threading option which is a feature that allows you to analyze the performance of a multi-threaded application. When analyzing a multi-threaded application, VTune Threading enables you to gather performance data from each thread separately and then aggregate the results to get a more comprehensive view of the application's performance.

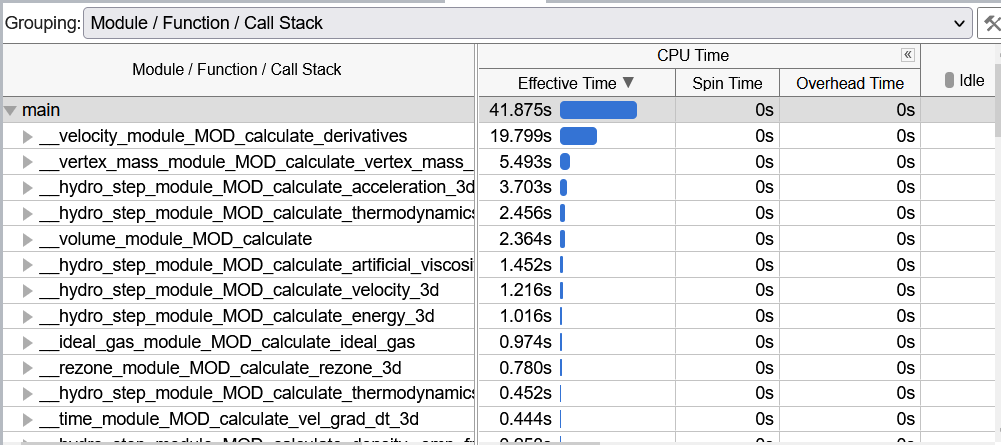
With this option, I’ve found places with hotspots, which are functions on the project, that were using the most time in the application and the exact duration spent in each function.

This information gave me the performance bottleneck in the code and where should I focus on.

From the profiler I’ve known that the best place to perform optimization was calculate\_derivatives, and a couple more functions which ill mention later in the report.

Each one of these hotspots that were found, I dealt with her with the openmp pragmas of fortran.

Picture from the Vtune showing the functions and the time each one spent on the run: (We can see that calculate derivatives took the most time, so we started by handling it first)

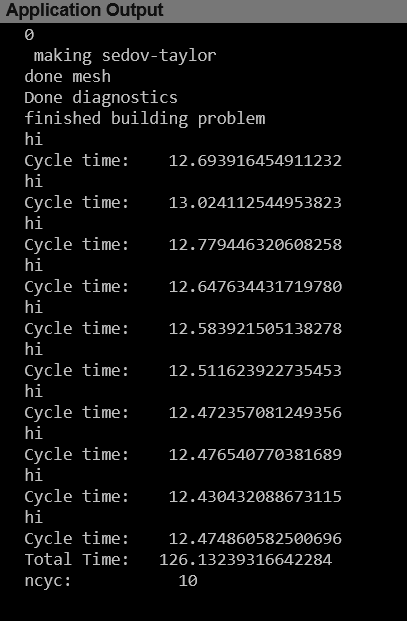


(I have to mention that the Vtune profiler didn’t work as I expected since I have the problem of seeing the function as there location on the code, assembly location, I informed Yehonathan about the issue.)

**My achievements:**

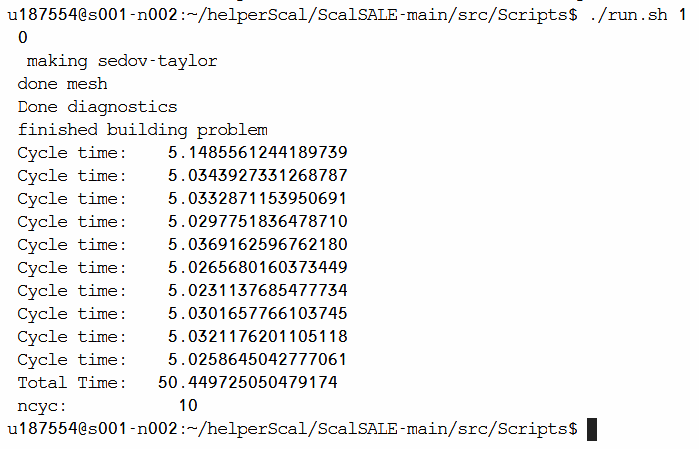
First thing I did was running the project on serial mode without any optimization, to check the amount of time the running would take, and I’ve got this:

(Sorry for the print(“hi”) that were in the middle)



After that, I’ve decided to use -o3 flag on the cmakelist, which is a compiler optimization flag that enables the highest level of optimization in the compiler.

And I’ve got the result of **50 seconds** total time:



Last thing, was doing the optimizations using the open-mp pragmas of fortran while keeping -o3 flag, and I’ve reached the total time of 10 second approx.

Text

Description automatically generated

**The places where I used OpenMp pragmas:**

To parallelize a specific part of my application using OpenMP, I first identified the section of code suitable for parallelization, which happened to be calculate\_derivatives in velocity.f90, and so on, based on the previous identification using Vtune.

After parallelizing first function (in velocity.f90), I recompiled the code, and ran Vtune again to check the accomplishments so far, and checked again what’s the next place to parallelize.  
I kept parallelizing function and compiling and checking again, until I saw no further optimization comes out of parallelizing more function, and there I stopped and that’s were the results:

We can see in the picture the places where the pragma “omp parallel do” was added, in each place of them there was a function that the vtune gave us, indicating that it ran a good amount of time and could be parallelized.



**Describe what compilers you chose to use:**

At first I searched google for what is better between the three compilers: ifort, gfortran and ifx, and I got this:

ifort is known for its high performance and optimization capabilities, particularly when running on Intel processors. This can make it a good choice for applications that require maximum performance, such as scientific or engineering simulations.

And ifort has good support for modern Fortran standards, which can make it easier to write and maintain high-quality Fortran code. It also has good support for OpenMP and MPI, which can be useful for parallelizing code on multi-core or distributed systems.

And after that I tested the three compilers on the second phase of parallelization (Where I had -O3 flag and no further openMP optimization), and I found that the three of them gave the same time, so given the explanation that I found on google, I’ve decided to go with iFort.

**GPU offloading:**

1. The program consists of a main loop (in problem.f90) which is dependent on previous iterations, making it difficult to offload to the GPU, To offload a function to the GPU, the longest function was chosen which was Calculate\_derivatives (which was found in the vtune before).
2. Offloading the entire program to the GPU would be not feasible for this exercise, so I didn’t do that, I only offloaded the calculate\_derivative.
3. The solution is to map the relevant data structures before the main loop, update them on each iteration, and transfer arrays from CPU to GPU before Calculate\_derivatives, and arrays from GPU to CPU after.

The implementation of this offloading was in two files, the problem.f90 and in velocity.f90, and all this was in the GPU folder.

I’ve added the relevant pragma and offloading function before the main loop in problem.f90, and add before and after the loop in velocity.f90 (calculate\_derivatives) to send the data and get it back and update…